WinDriver USB v8.02 User's Guide

Jungo Ltd

COPYRIGHT

Contents

List of Figures

Chapter 1

WinDriver Overview

In this chapter you will explore the uses of WinDriver, and learn the basic steps of creating your driver.
The WinDriver USB Device toolkit, for development of USB device firmware code, is outlined separately in Chapter 12.

> **NOTE**
> This manual outlines WinDriver's support for USB devices. WinDriver also supports development for PCI/PCMCIA/CardBus/ISA/EISA/CompactPCI/PCI Express devices. For detailed information regarding WinDriver's support for these buses, please refer to the WinDriver Product Line page on our web-site (`http://www.jungo.com/windriver.html`) and to the WinDriver PCI/PCMCIA/CardBus/ISA/EISA/CompactPCI/PCI Express User's Manual, which is available on-line at:
> `http://www.jungo.com/support/support_windriver.html`.
>
> Support for USB on Windows NT 4.0 is provided in a separate tool-kit – see our WinDriver USB for NT web-page: `http://www.jungo.com/wdusb_nt.html`.

1.1 Introduction to WinDriver

WinDriver is a development toolkit that dramatically simplifies the difficult task of creating device drivers and hardware access applications. WinDriver includes a wizard and code generation features that automatically detect your hardware and generate the driver to access it from your application. The driver and application you develop using WinDriver is source code compatible between all supported operating

systems (WinDriver currently supports Windows 98/Me/2000/XP/Server2003, Windows CE.NET/Windows Mobile 5.0 and Linux). The driver is binary compatible between Windows 98/Me/2000/XP/Server 2003. WinDriver provides a complete solution for creating high-performance drivers.

Don't let the size of this manual fool you. WinDriver makes developing device drivers an easy task that takes hours instead of months. Most of this manual deals with the features that WinDriver offers to the advanced user. However, most developers will find that reading this chapter and glancing through the DriverWizard and function reference chapters is all they need to successfully write their driver.

WinDriver supports development for all USB chipsets. Enhanced support is offered for Cypress, Microchip, Philips, Texas Instruments and Silicon Laboratories USB chipsets, as outlined in Chapter 8 of the manual.

Visit Jungo's web site at `http://www.jungo.com` for the latest news about WinDriver and other driver development tools that Jungo offers.

1.2 Background

1.2.1 The Challenge

In protected operating systems such as Windows and Linux, a programmer cannot access hardware directly from the application level (user mode), where development work is usually done. Hardware can only be accessed from within the operating system itself (kernel mode or Ring-0), utilizing software modules called device drivers. In order to access a custom hardware device from the application level, a programmer must do the following:

- Learn the internals of the operating system he is working on (Windows 98/Me/2000/XP/Server2003, Windows CE.NET/Windows Mobile 5.0 and Linux).
- Learn how to write a device driver.
- Learn new tools for developing/debugging in kernel mode (DDK, ETK, DDI/DKI).
- Write the kernel-mode device driver that does the basic hardware input/output.
- Write the application in user mode that accesses the hardware through the device driver written in kernel mode.
- Repeat the first four steps for each new operating system on which the code should run.

1.2.2 The WinDriver Solution

Easy Development: WinDriver enables Windows 98 / Me / 2000 / XP / Server2003, Windows CE.NET / Windows Mobile 5.0 and Linux programmers to create USB based device drivers in an extremely short time. WinDriver allows you to create your driver in the familiar user-mode environment, using MSDEV/Visual C/C++, MSDEV .NET, Borland C++ Builder, Borland Delphi, Visual Basic 6.0, MS eMbedded Visual C++, MS Platform Builder C++, GCC, or any other appropriate compiler. You do not need to have any device driver knowledge, nor do you have to be familiar with operating system internals, kernel programming, the DDK, ETK or DDI/DKI.

Cross Platform: The driver created with WinDriver will run on Windows 98/Me/2000/XP/Server2003, Windows CE.NET/Windows Mobile 5.0 and Linux. In other words – write it once, run it on many platforms.

Friendly Wizards: DriverWizard (included) is a graphical diagnostics tool that lets you view the device's resources and test the communication with the hardware with just a few mouse clicks, before writing a single line of code. Once the device is operating to your satisfaction, DriverWizard creates the skeletal driver source code, giving access functions to all the resources on the hardware.

Kernel-Mode Performance: WinDriver's API is optimized for performance.

1.3 Conclusion

Using WinDriver, a developer need only do the following to create an application that accesses the custom hardware:

- Start DriverWizard and detect the hardware and its resources.
- Automatically generate the device driver code from within DriverWizard, or use one of the WinDriver samples as the basis for the application (see Chapter 8 for an overview of WinDriver's enhanced support for specific chipsets).
- Modify the user-mode application, as needed, using the generated/sample functions to implement the desired functionality for your application.

Your hardware access application will run on all the supported platforms: Windows 98/Me/2000/XP/Server2003, Windows CE.NET/Windows Mobile 5.0 and Linux – just re-compile the code for the target platform. (The code is binary compatible between Windows 98/Me/2000/XP/Server 2003 platforms, so there is no need to rebuild the code when porting the driver between these operating systems.)

1.4 WinDriver Benefits

- Easy user-mode driver development.
- Friendly DriverWizard allows hardware diagnostics without writing a single line of code.
- Automatically generates the driver code for the project in C, C#, Visual Basic .NET, Delphi (Pascal) or Visual Basic.
- Supports any USB device, regardless of manufacturer.
- Enhanced support for Cypress, Microchip, Philips, Texas Instruments and Silicon Laboratories chipsets frees the developer from the need to study the hardware's specification.
- Applications are binary-compatible across Windows 98 / Me / 2000 / XP / Server 2003.
- Applications are source code compatible across Windows 98 / Me / 2000 / XP / Server2003, Windows CE.NET / Windows Mobile 5.0 and Linux.
- Can be used with common development environments, including MSDEV/Visual C/C++, MSDEV .NET, Borland C++ Builder, Borland Delphi, Visual Basic 6.0, MS eMbedded Visual C++, MS Platform Builder C++, GCC, or any other appropriate compiler.
- No DDK, ETK, DDI or any system-level programming knowledge required.
- Supports multiple CPUs.
- Includes dynamic driver loader.
- Comprehensive documentation and help files.
- Detailed examples in C, C#, Visual Basic .NET, Delphi and Visual Basic 6.0.
- WHQL certifiable driver (Windows).
- Two months of free technical support.
- No run-time fees or royalties.

1.5 WinDriver Architecture

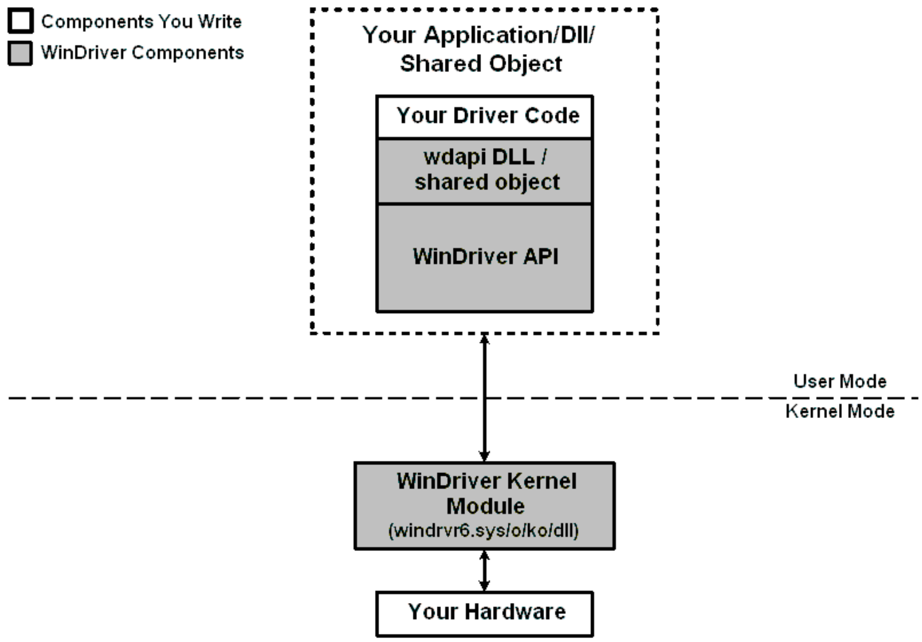


Figure 1.1: WinDriver Architecture

For hardware access, your application calls one of the WinDriver user-mode functions. The user-mode function calls the WinDriver kernel, which accesses the hardware for you through the native calls of the operating system.

1.6 What Platforms Does WinDriver Support?

WinDriver supports Windows 98/Me/2000/XP/Server2003, Windows CE.NET/Windows Mobile 5.0 and Linux.

The same source code will run on all supported platforms – simply re-compile it for the target platform. The source code is binary compatible across Windows 98/Me/2000/XP/Server 2003, so executables created with WinDriver can be ported between these operating systems without re-compilation.

Even if your code is meant only for one of the supported operating systems, using WinDriver will give you the flexibility to move your driver to another operating system in the future without needing to change your code.

1.7 Limitations of the Different Evaluation Versions

All the evaluation versions of the WinDriver USB Host toolkit are full featured. No functions are limited or crippled in any way. The evaluation version of WinDriver varies from the registered version in the following ways:

- Each time WinDriver is activated, an **Un-registered** message appears.
- When using the DriverWizard, a dialogue box with a message stating that an evaluation version is being run appears on every interaction with the hardware.
- In the Linux and CE versions, the driver will remain operational for 60 minutes, after which time it must be restarted.
- The Windows evaluation version expires 30 days from the date of installation.

For more details please refer to appendix G.

1.8 How Do I Develop My Driver with WinDriver?

1.8.1 On Windows 98/Me/2000/XP/Server 2003 and Linux

1. Start DriverWizard and use it to diagnose your hardware – see details in Chapter 5.
2. Let DriverWizard generate skeletal code for your driver, or use one of the WinDriver samples as the basis for your driver application (see Chapter [8] for details regarding WinDriver's enhanced support for specific chipsets).
3. Modify the generated/sample code to suit your application's needs.
4. Run and debug your driver.

NOTE
The code generated by DriverWizard is a diagnostics program that contains functions that perform data transfers on the device's pipes, send requests to the control pipe, change the active alternate setting, reset pipes, and more.

1.8.2 On Windows CE

1. Plug your hardware into a Windows host machine.
2. Diagnose your hardware using DriverWizard.
3. Let DriverWizard generate your driver's skeletal code.
4. Modify this code using eMbedded Visual C++ to meet your specific needs. If you are using Platform Builder, activate it and insert the generated ***.pbp** into your workspace.
5. Test and debug your code and hardware from the CE emulation running on the host machine.

1.9 What Does the WinDriver Toolkit Include?

- A printed version of this manual
- Two months of free technical support (Phone/Fax/Email)
- WinDriver modules
- The WinDriver CD
  - Utilities
  - Chipset support APIs
  - Sample files

1.9.1 WinDriver Modules

- WinDriver (**WinDriver/include**) – the general purpose hardware access toolkit. The main files here are:
  - **windrvr.h**: Declarations and definitions of WinDriver's basic API.
  - **wdu_lib.h**: Declarations and definitions of the WinDriver USB (WDU) library, which provides convenient wrapper USB APIs.
  - **windrvr_int_thread.h**: Declarations of convenient wrapper functions to simplify interrupt handling.
  - **windrvr_events.h**: Declarations of APIs for handling and Plug-and-Play and power management events.
  - **utils.h**: Declarations of general utility functions.

  - **status_strings.h**: Declarations of API for converting WinDriver status codes to descriptive error strings.

- DriverWizard (**WinDriver/wizard/wdwizard**) – a graphical tool that diagnoses your hardware and enables you to easily generate code for your driver (refer to Chapter 5 for details).
- Graphical Debugger (**WinDriver/util/wddebug_gui**) – a graphical debugging tool that collects information about your driver as it runs.
  WinDriver also includes a console version of this program (**WinDriver/util/wddebug**), which can be used on platforms that have no GUI support, such as Windows CE .
  For details regarding the Debug Monitor, refer to section 7.2.
- WinDriver distribution package (**WinDriver/redist**) – the files you include in the driver distribution to customers.
- This manual – the full WinDriver manual (this document), in different formats, can be found under the **WinDriver/docs** directory.

1.9.2 Utilities

- **usb_diag.exe** (**WinDriver/util/usb_diag.exe**) – enables the user to view the resources of connected USB devices and communicate with the devices – transfer data to/from the device, set the active alternate setting, reset pipes, etc. On Windows 98/Me/2000/XP/Server 2003 the program identifies all devices that have been registered to work with WinDriver using an INF file. On the other supported operating systems the program identifies all USB devices connected to the target platform.
- **pci_dump.exe** (**WinDriver/util/pci_dump.exe**) – used to obtain a dump of the PCI configuration registers of the installed PCI cards.

The Windows CE version also includes:

- **\REDIST\... \X86EMU\WINDRVR_CE_EMU.DLL**: DLL that communicates with the WinDriver kernel – for the x86 HPC emulation mode of Windows CE.
- **\REDIST\... \X86EMU\WINDRVR_CE_EMU.LIB:** an import library that is used to link with WinDriver applications that are compiled for the x86 HPC emulation mode of Windows CE.

1.9.3 WinDriver's Specific Chipset Support

WinDriver provides custom wrapper APIs and sample code for major USB chipsets (see Chapter 8), including for the following chipsets:

- Cypress EZ-USB – **WinDriver/cypress**
- Microchip PIC18F4550 – **WinDriver/microchip/pic18f4550**
- Philips PDIUSBD12 – **WinDriver/pdiusbd12**
- Texas Instruments TUSB3410, TUSB3210, TUSB2136 and TUSB5052 – **WinDriver/ti**
- Silicon Laboratories C8051F320 USB – **WinDriver/silabs**

1.9.4 Samples

In addition to the samples provided for specific chipsets [1.9.3], WinDriver includes a variety of samples that demonstrate how to use WinDriver's API to communicate with your device and perform various driver tasks.

- C samples: found under the **WinDriver/samples** directory.
  These samples also include the source code for the utilities listed above [1.9.2].
- .NET C# and Visual Basic .NET samples (Windows): found under the **WinDriver****csharp.net** and **WinDriver****vb.net** directories (respectively).
- Delphi (Pascal) samples (Windows) **WinDriver****delphi****samples** directory.
- Visual Basic samples (Windows): found under the **WinDriver****vb****samples** directory.

1.10 Can I Distribute the Driver Created with WinDriver?

Yes. WinDriver is purchased as a development toolkit, and any device driver created using WinDriver may be distributed, royalties free, in as many copies as you wish. See the license agreement (**WinDriver/docs/license.pdf**) for more details.

Chapter 2

Understanding Device Drivers

This chapter provides you with a general introduction to device drivers and takes you through the structural elements of a device driver.

> **NOTE**
> Using WinDriver, you do not need to familiarize yourself with the internal workings of driver development. As explained in Chapter 1 of the manual, WinDriver enables you to communicate with your hardware and develop a driver for your device from the user mode, using only WinDriver's simple APIs, without any need for driver or kernel development knowledge.

2.1 Device Driver Overview

Device drivers are the software segments that provides an interface between the operating system and the specific hardware devices such as terminals, disks, tape drives, video cards and network media. The device driver brings the device into and out of service, sets hardware parameters in the device, transmits data from the kernel to the device, receives data from the device and passes it back to the kernel, and handles device errors.

A driver acts like a translator between the device and programs that use the device. Each device has its own set of specialized commands that only its driver knows. In contrast, most programs access devices by using generic commands. The driver, therefore, accepts generic commands from a program and then translates them into specialized commands for the device.

2.2 Classification of Drivers According to Functionality

There are numerous driver types, differing in their functionality. This subsection briefly describes three of the most common driver types.

2.2.1 Monolithic Drivers

Monolithic drivers are device drivers that embody all the functionality needed to support a hardware device. A monolithic driver is accessed by one or more user applications, and directly drives a hardware device. The driver communicates with the application through I/O control commands (IOCTLs) and drives the hardware using calls to the different DDK, ETK, DDI/DKI functions.

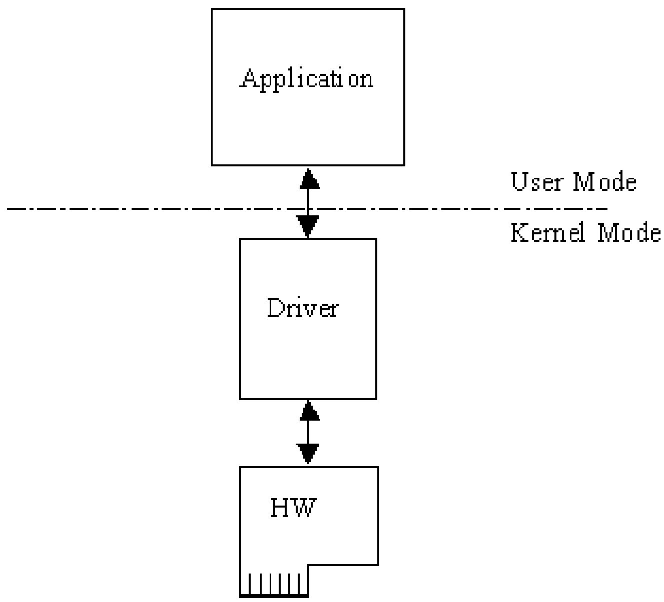


Figure 2.1: Monolithic Drivers

Monolithic drivers are supported in all operating systems including all Windows platforms and all Unix platforms.

2.2.2 Layered Drivers

Layered drivers are device drivers that are part of a stack of device drivers that together process an I/O request. An example of a layered driver is a driver that intercepts calls to the disk and encrypts/decrypts all data being transferred to/from the disk. In this example, a driver would be hooked on to the top of the existing driver and would only do the encryption/decryption.

Layered drivers are sometimes also known as filter drivers, and are supported in all operating systems including all Windows platforms and all Unix platforms.

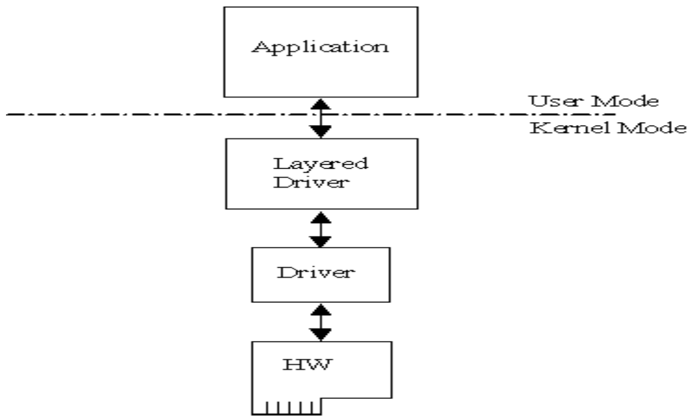


Figure 2.2: Layered Drivers

2.2.3 Miniport Drivers

A Miniport driver is an add-on to a class driver that supports miniport drivers. It is used so the miniport driver does not have to implement all of the functions required of a driver for that class. The class driver provides the basic class functionality for the miniport driver.
A class driver is a driver that supports a group of devices of common functionality, such as all HID devices or all network devices.

Miniport drivers are also called miniclass drivers or minidrivers, and are supported in the Windows NT (or 2000) family, namely Windows NT/2000/XP and Server 2003.

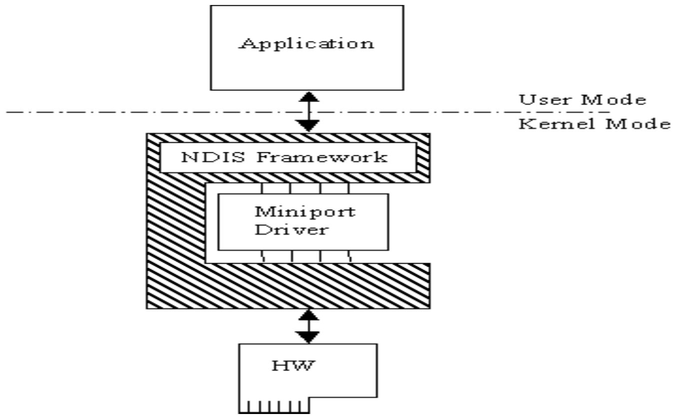


Figure 2.3: Miniport Drivers

Windows NT/2000/XP/Server 2003 provide several driver classes (called ports) that handle the common functionality of their class. It is then up to the user to add only the functionality that has to do with the inner workings of the specific hardware. The NDIS miniport driver is one example of such a driver. The NDIS miniport framework is used to create network drivers that hook up to NT's communication stacks, and are therefore accessible to common communication calls used by applications. The Windows NT kernel provides drivers for the various communication stacks and other code that is common to communication cards. Due to the NDIS framework, the network card developer does not have to write all of this code, only the code that is specific to the network card he is developing.

2.3 Classification of Drivers According to Operating Systems

2.3.1 WDM Drivers

WDM (Windows Driver Model) drivers are kernel-mode drivers within the Windows NT and Windows 98 operating system families. Windows NT family includes Windows NT/2000/XP/Server 2003, and Windows 98 family includes Windows 98 and Windows Me.

WDM works by channeling some of the work of the device driver into portions of the code that are integrated into the operating system. These portions of code handle all of the low-level buffer management, including DMA and Plug and Play (Pnp) device enumeration.

WDM drivers are PnP drivers that support power management protocols, and include monolithic drivers, layered drivers and miniport drivers.

2.3.2 VxD Drivers

VxD drivers are Windows 95/98/Me Virtual Device Drivers, often called VxDs because the file names end with the .vxd extension. VxD drivers are typically monolithic in nature. They provide direct access to hardware and privileged operating system functions. VxD drivers can be stacked or layered in any fashion, but the driver structure itself does not impose any layering.

2.3.3 Unix Device Drivers

In the classic Unix driver model, devices belong to one of three categories: character (char) devices, block devices and network devices. Drivers that implement these devices are correspondingly known as char drivers, block drivers or network drivers. Under Unix, drivers are code units linked into the kernel that run in privileged kernel mode. Generally, driver code runs on behalf of a user-mode application. Access to Unix drivers from user-mode applications is provided via the file system. In other words, devices appear to the applications as special device files that can be opened.

Unix device drivers are either layered or monolithic drivers. A monolithic driver can be perceived as a one-layer layered driver.

2.3.4 Linux Device Drivers

Linux device drivers are based on the classic Unix device driver model. In addition, Linux introduces some new characteristics.

Under Linux, a block device can be accessed like a character device, as in Unix, but also has a block-oriented interface that is invisible to the user or application.

Traditionally, under Unix, device drivers are linked with the kernel, and the system is brought down and restarted after installing a new driver. Linux introduces the concept of a dynamically loadable driver called a module. Linux modules can be loaded or removed dynamically without requiring the system to be shut down. A Linux driver can be written so that it is statically linked or written in a modular form that allows it to be dynamically loaded. This makes Linux memory usage very efficient because modules can be written to probe for their own hardware and unload themselves if they cannot find the hardware they are looking for.

Like Unix device drivers, Linux device drivers are either layered or monolithic drivers.

2.4 The Entry Point of the Driver

Every device driver must have one main entry point, like the `main()` function in a C console application. This entry point is called `DriverEntry()` in Windows and `init_module()` in Linux. When the operating system loads the device driver, this driver entry procedure is called.

There is some global initialization that every driver needs to perform only once when it is loaded for the first time. This global initialization is the responsibility of the `DriverEntry()`/`init_module()` routine. The entry function also registers which driver callbacks will be called by the operating system. These driver callbacks are operating system requests for services from the driver. In Windows, these callbacks are called *dispatch routines*, and in Linux they are called *file operations*. Each registered callback is called by the operating system as a result of some criteria, such as disconnection of hardware, for example.

2.5 Associating the Hardware to the Driver

Operating systems differ in how they link a device to its driver.
In Windows, the link is performed by the INF file, which registers the device to work with the driver. This association is performed before the `DriverEntry()` routine is called. The operating system recognizes the device, looks up in its database which INF file is associated with the device, and according to the INF file, calls the driver's entry point.

In Linux, the link between a device and its driver is defined in the `init_module()` routine. The `init_module()` routine includes a callback which states what hardware the driver is designated to handle. The operating system calls the driver's entry point, based on the definition in the code.

2.6 Communicating with Drivers

A driver can create an instance, thus enabling an application to open a handle to the driver through which the application can communicate with it.
The applications communicate with the drivers using a file access API (Application Program Interface). Applications open a handle to the driver using `CreateFile()` call (in Windows), or `open()` call (in Linux) with the name of the device as the file name. In order to read from and write to the device, the application calls `ReadFile()` and `WriteFile()` (in Windows), or `read()`, `write()` in Linux.

Sending requests is accomplished using an I/O control call, called `DeviceIoControl()` (in Windows), and `ioctl()` in Linux. In this I/O control call, the application specifies:

- The device to which the call is made (by providing the device's handle).
- An IOCTL code that describes which function this device should perform.
- A buffer with the data on which the request should be performed.

The IOCTL code is a number that the driver and the requester agree upon for a common task.

The data passed between the driver and the application is encapsulated into a structure. In Windows, this structure is called an I/O Request Packet (IRP), and is encapsulated by the I/O Manager. This structure is passed on to the device driver, which may modify it and pass it down to other device drivers.

Chapter 3

WinDriver USB Overview

This chapter explores the basic characteristics of the Universal Serial Bus (USB) and introduces WinDriver USB's features and architecture.

> **NOTE**
> The references to the WinDriver USB toolkit in this chapter relate to the standard WinDriver USB toolkit for development of USB host drivers.
> The WinDriver USB Device toolkit, designed for development of USB device firmware, is discussed separately in Chapter 12.

3.1 Introduction to USB

USB (Universal Serial Bus) is an industry standard extension to the PC architecture for attaching peripherals to the computer. It was originally developed in 1995 by leading PC and telecommunication industry companies, such as Intel, Compaq, Microsoft and NEC. USB was developed to meet several needs, among them the needs for an inexpensive and widespread connectivity solution for peripherals in general and for computer telephony integration in particular, an easy-to-use and flexible method of reconfiguring the PC, and a solution for adding a large number of external peripherals. The USB standard meets these needs.

The USB specification allows for the connection of a maximum of 127 peripheral devices (including hubs) to the system, either on the same port or on different ports.

USB also supports Plug and Play installation and hot swapping.

The **USB 1.1** standard supports both isochronous and asynchronous data transfers and has dual speed data transfer: 1.5 Mb/s (megabits per second) for **low-speed** USB

devices and 12 Mb/s for **high-speed** USB devices (much faster than the original serial port). Cables connecting the device to the PC can be up to five meters (16.4 feet) long. USB includes built-in power distribution for low power devices and can provide limited power (up to 500 mA of current) to devices attached on the bus.

The **USB 2.0** standard supports a signalling rate of 480 Mb/s, known as **”high-speed”**, which is 40 times faster than the USB 1.1 full-speed transfer rate.
USB 2.0 is fully forward- and backward-compatible with USB 1.1 and uses existing cables and connectors.
USB 2.0 supports connections with PC peripherals that provide expanded functionality and require wider bandwidth. In addition, it can handle a larger number of peripherals simultaneously.
USB 2.0 enhances the user’s experience of many applications, including interactive gaming, broadband Internet access, desktop and Web publishing, Internet services and conferencing.

Because of its benefits (described also in section 3.2 below), USB is currently enjoying broad market acceptance.

3.2 WinDriver USB Benefits

This section describes the main benefits of the USB standard and the WinDriver USB toolkit, which supports this standard:

- External connection, maximizing ease of use
- Self identifying peripherals supporting automatic mapping of function to driver and configuration
- Dynamically attachable and re-configurable peripherals
- Suitable for device bandwidths ranging from a few Kb/s to hundreds of Mb/s
- Supports isochronous as well as asynchronous transfer types over the same set of wires
- Supports simultaneous operation of many devices (multiple connections)
- Supports a data transfer rate of up to 480 Mb/s (high-speed) for USB 2.0 (for the operating systems that officially support this standard) and up to 12 Mb/s (full-speed) for USB 1.1
- Guaranteed bandwidth and low latencies; appropriate for telephony, audio, etc. (isochronous transfer may use almost the entire bus bandwidth)
- Flexibility: supports a wide range of packet sizes and a wide range of data transfer rates

- Robustness: built-in error handling mechanism and dynamic insertion and removal of devices with no delay observed by the user
- Synergy with PC industry; Uses commodity technologies
- Optimized for integration in peripheral and host hardware
- Low-cost implementation, therefore suitable for development of low-cost peripherals
- Low-cost cables and connectors
- Built-in power management and distribution

3.3 USB Components

The Universal Serial bus is comprised of the following primary components:

USB Host: The USB host platform is where the USB host controller is installed and where the client software/device driver runs. The *USB Host Controller* is the interface between the host and the USB peripherals. The host is responsible for detecting the insertion and removal of USB devices, managing the control and data flow between the host and the devices, providing power to attached devices and more.

USB Hub: A USB device that allows multiple USB devices to attach to a single USB port on a USB host. Hubs on the back plane of the hosts are called *root hubs*. Other hubs are called *external hubs*.

USB Function: A USB device that can transmit or receive data or control information over the bus and that provides a function. A function is typically implemented as a separate peripheral device that plugs into a port on a hub using a cable. However, it is also possible to create a *compound device*, which is a physical package that implements multiple functions and an embedded hub with a single USB cable. A compound device appears to the host as a hub with one or more non-removable USB devices, which may have ports to support the connection of external devices.

3.4 Data Flow in USB Devices

During the operation of a USB device, the host can initiate a flow of data between the client software and the device.

Data can be transferred between the host and only one device at a time (*peer to peer communication*). However, two hosts cannot communicate directly, nor can two USB

devices (with the exception of On-The-Go (OTG) devices, where one device acts as the master (host) and the other as the slave.)

The data on the USB bus is transferred via pipes that run between software memory buffers on the host and endpoints on the device.

Data flow on the USB bus is half-duplex, i.e. data can be transmitted only in one direction at a given time.

An **endpoint** is a uniquely identifiable entity on a USB device, which is the source or terminus of the data that flows from or to the device. Each USB device, logical or physical, has a collection of independent endpoints. The three USB speeds (low, full and high) all support one bi-directional control endpoint (endpoint zero) and 15 unidirectional endpoints. Each unidirectional endpoint can be used for either inbound or outbound transfers, so theoretically there are 30 supported endpoints.
Each endpoint has the following attributes: bus access frequency, bandwidth requirement, endpoint number, error handling mechanism, maximum packet size that can be transmitted or received, transfer type and direction (into or out of the device).

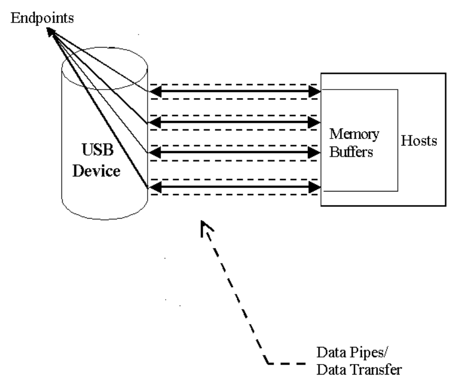


Figure 3.1: USB Endpoints

A **pipe** is a logical component that represents an association between an endpoint on the USB device and software on the host. Data is moved to and from a device through a pipe. A pipe can be either a stream pipe or a message pipe, depending on the type of data transfer used in the pipe. *Stream pipes* handle interrupt, bulk and isochronous transfers, while *message pipes* support the control transfer type. The different USB transfer types are discussed below [3.6].

3.5 USB Data Exchange

The USB standard supports two kinds of data exchange between a host and a device: functional data exchange and control exchange.

Functional data exchange is used to move data to and from the device. There are three types of data transfers: bulk, interrupt and isochronous.

Control exchange is used to determine device identification and configuration requirements and to configure a device, and can also be used for other device-specific purposes, including control of other pipes on the device. Control exchange takes place via a control pipe, mainly the default *Pipe 0*, which always exists. The control transfer consists of a *setup stage* (in which a setup packet is sent from the host to the device), an optional *data stage* and a *status stage*.

Figure 3.2 below depicts a USB device with one bi-directional control pipe (endpoint) and six functional data transfer pipes (endpoints), as identified by WinDriver's DriverWizard utility (discussed in Chapter 5).

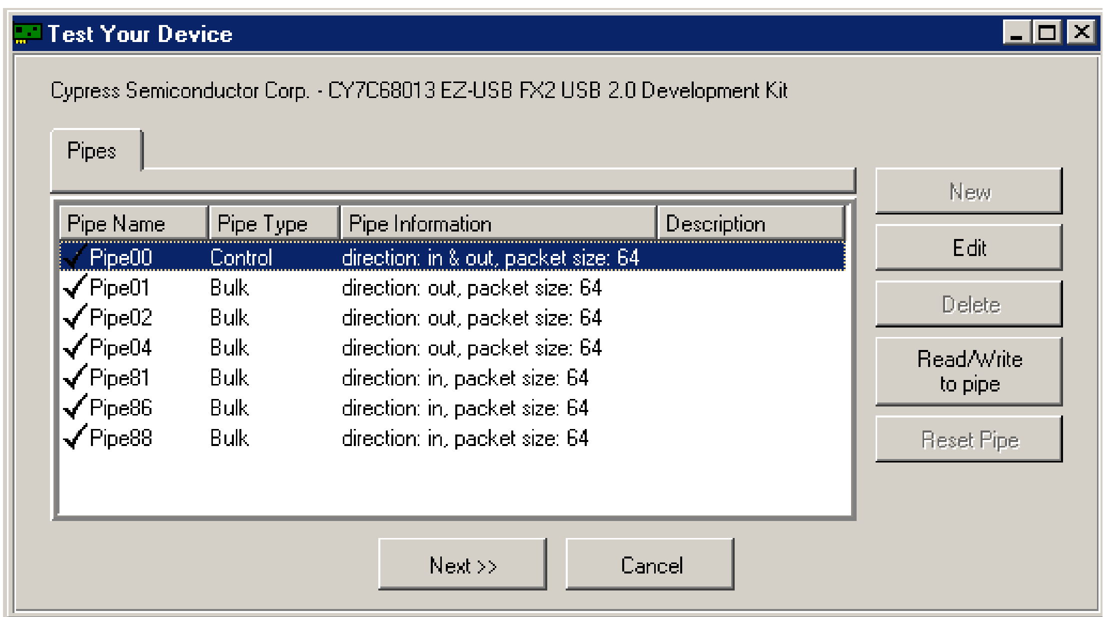


Figure 3.2: USB Pipes

More information on how to implement the control transfer by sending setup packets can be found in Chapter 9.

3.6 USB Data Transfer Types

The USB device (function) communicates with the host by transferring data through a pipe between a memory buffer on the host and an endpoint on the device. USB supports four different transfer types. A type is selected for a specific endpoint according to the requirements of the device and the software. The transfer type of a specific endpoint is determined in the endpoint descriptor.

The USB specification provides for the following data transfer types:

3.6.1 Control Transfer

Control Transfer is mainly intended to support configuration, command and status operations between the software on the host and the device.

This transfer type is used for low-, full- and high-speed devices.

Each USB device has at least one control pipe (default pipe), which provides access to the configuration, status and control information.

Control transfer is bursty, non-periodic communication.

The control pipe is bi-directional – i.e. data can flow in both directions.

Control transfer has a robust error detection, recovery and retransmission mechanism and retries are made without the involvement of the driver.

The maximum packet size for control endpoints can be only 8 bytes for low-speed devices; 8, 16, 32, or 64 bytes for full-speed devices; and only 64 bytes for high-speed devices.

3.6.2 Isochronous Transfer

Isochronous Transfer is most commonly used for time-dependent information, such as multimedia streams and telephony.

This transfer type can be used by full-speed and high-speed devices, but not by low-speed devices.

Isochronous transfer is periodic and continuous.

The isochronous pipe is unidirectional, i.e. a certain endpoint can either transmit or receive information. Bi-directional isochronous communication requires two isochronous pipes, one in each direction.

USB guarantees the isochronous transfer access to the USB bandwidth (i.e. it reserves the required amount of bytes of the USB frame) with bounded latency, and guarantees the data transfer rate through the pipe, unless there is less data transmitted.

Since timeliness is more important than correctness in this type of transfer, no retries are made in case of error in the data transfer. However, the data receiver can determine that an error occurred on the bus.

3.6.3 Interrupt Transfer

Interrupt Transfer is intended for devices that send and receive small amounts of data infrequently or in an asynchronous time frame.

This transfer type can be used for low-, full- and high-speed devices.

Interrupt transfer type guarantees a maximum service period and that delivery will be re-attempted in the next period if there is an error on the bus.

The interrupt pipe, like the isochronous pipe, is unidirectional and periodical.

The maximum packet size for interrupt endpoints can be 8 bytes or less for low-speed devices; 64 bytes or less for full-speed devices; and 1,024 bytes or less for high-speed devices.

3.6.4 Bulk Transfer

Bulk Transfer is typically used for devices that transfer large amounts of non-time sensitive data, and that can use any available bandwidth, such as printers and scanners.

This transfer type can be used by full-speed and high-speed devices, but not by low-speed devices.

Bulk transfer is non-periodic, large packet, bursty communication.

Bulk transfer allows access to the bus on an "as-available" basis, guarantees the data transfer but not the latency, and provides an error check mechanism with retries attempts. If part of the USB bandwidth is not being used for other transfers, the system will use it for bulk transfer.

Like the other stream pipes (isochronous and interrupt), the bulk pipe is also unidirectional, so bi-directional transfers require two endpoints.

The maximum packet size for bulk endpoints can be 8, 16, 32, or 64 bytes for full-speed devices, and 512 bytes for high-speed devices.

3.7 USB Configuration

Before the USB function (or functions, in a compound device) can be operated, the device must be configured. The host does the configuring by acquiring the configuration information from the USB device. USB devices report their attributes by descriptors. A **descriptor** is the defined structure and format in which the data is transferred. A complete description of the USB descriptors can be found in Chapter 9 of the USB Specification (see http://www.usb.org for the full specification).

It is best to view the USB descriptors as a hierarchical structure with four levels:

- The *Device* level
- The *Configuration* level
- The *Interface* level (this level may include an optional sub-level called *Alternate Setting*)
- The *Endpoint* level

There is only one device descriptor for each USB device. Each device has one or more configurations, each configuration has one or more interfaces, and each interface has zero or more endpoints, as demonstrated in Figure 3.3 below.

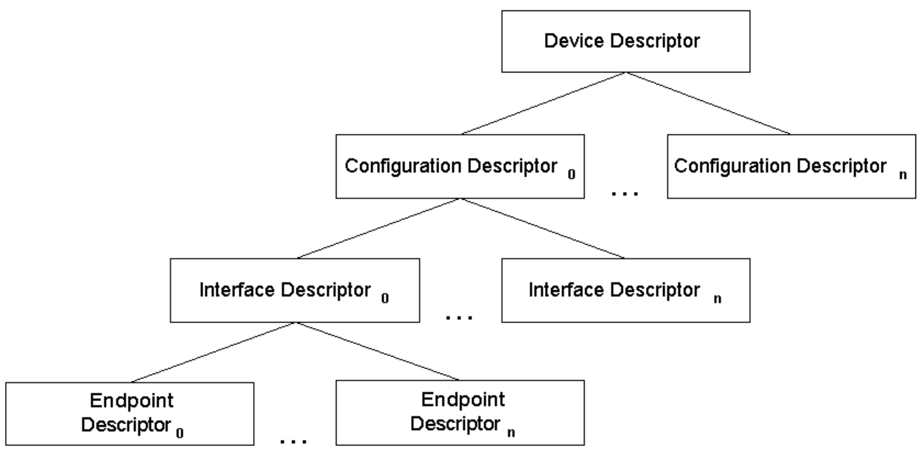


Figure 3.3: Device Descriptors

Device Level: The device descriptor includes general information about the USB device, i.e. global information for all of the device configurations. The device descriptor identifies, among other things, the device class (HID device, hub, locator device, etc.), subclass, protocol code, vendor ID, device ID and more. Each USB device has one device descriptor.

Configuration Level: A USB device has one or more configuration descriptors. Each descriptor identifies the number of interfaces grouped in the configuration and the power attributes of the configuration (such as self-powered, remote wakeup, maximum power consumption and more). Only one configuration can be loaded at a given time. For example, an ISDN adapter might have two different configurations, one that presents it with a single interface of 128 Kb/s and a second that presents it with two interfaces of 64 Kb/s each.

Interface Level: The interface is a related set of endpoints that present a specific functionality or feature of the device. Each interface may operate independently. The interface descriptor describes the number of the interface, the number of endpoints used by this interface and the interface-specific class, subclass and protocol values when the interface operates independently.

In addition, an interface may have **alternate settings**. The alternate settings allow the endpoints or their characteristics to be varied after the device is configured.

Endpoint Level: The lowest level is the endpoint descriptor, which provides the host with information regarding the endpoint's data transfer type and maximum packet size. For isochronous endpoints, the maximum packet size is used to reserve the required bus time for the data transfer – i.e. the bandwidth. Other endpoint attributes are its bus access frequency, endpoint number, error handling mechanism and direction.

The same endpoint can have different properties (and consequently different uses) in different alternate settings.

Seems complicated? Not at all! WinDriver automates the USB configuration process. The included DriverWizard utility [5] and USB diagnostics application scan the USB bus, detect all USB devices and their configurations, interfaces, alternate settings and endpoints, and enable you to pick the desired configuration before starting driver development.

WinDriver identifies the endpoint transfer type as determined in the endpoint descriptor. The driver created with WinDriver contains all configuration information acquired at this early stage.

3.8 WinDriver USB

WinDriver USB enables developers to quickly develop high-performance drivers for USB-based devices, without having to learn the USB specifications or the operating system's internals.

Using WinDriver USB, developers can create USB drivers without having to use the operating system's development kits (such as the Windows DDK); In addition, Windows developers do not need to familiarize themselves with Microsoft's Win32 Driver Module (WDM).

The driver code developed with WinDriver USB is binary compatible across the supported Windows platforms – Windows 98/Me/2000/XP/Server 2003 – and source code compatible across all supported operating systems – Windows 98/Me/2000/XP/Server2003, Windows CE.NET/Windows Mobile 5.0 and Linux. For an up-to-date list of supported operating systems, visit Jungo's web site at: `http://www.jungo.com`.

WinDriver USB is a generic tool kit that supports all USB devices from all vendors and with all types of configurations.

WinDriver USB encapsulates the USB specification and architecture, letting you focus on your application logic. WinDriver USB features the graphical DriverWizard utility [5], which enables you to easily detect your hardware, view its configuration information, and test it, before writing a single line of code: DriverWizard first lets you choose the desired configuration, interface and alternate setting combination, using a friendly graphical user interface. After detecting and configuring your USB device, you can proceed to test the communication with the device – perform data transfers on the pipes, send control requests, reset the pipes, etc. – in order to ensure that all your hardware resources function as expected.

After your hardware is diagnosed, you can use DriverWizard to automatically generate your device driver source code in C, C#, Visual Basic .NET, Delphi or Visual Basic. WinDriver USB provides user-mode APIs, which you can call from within your application in order to implement the communication with your device. The WinDriver USB API includes USB-unique operations such as reset of a pipe or a device. The generated DriverWizard code implements a diagnostics application, which demonstrates how to use WinDriver's USB API to drive your specific device. In order to use the application you just need to compile and run it. You can jump-start your development cycle by using this application as your skeletal driver and then modifying the code, as needed, to implement the desired driver functionality for your specific device.

DriverWizard also automates the creation of an INF file that registers your device to work with WinDriver, which is an essential step in order to correctly identify and

handle USB devices using WinDriver. For an explanation on why you need to create an INF file for your USB device, refer to section 11.3.1 of the manual. For detailed information on creation of INF files with DriverWizard, refer to section 5.2 (see specifically step 3).

With WinDriver USB, all development is done in the user mode, using familiar development and debugging tools and your favorite compiler (such as MSDEV/Visual C/C++, MSDEV .NET, Borland C++ Builder, Borland Delphi, Visual Basic 6.0, MS eMbedded Visual C++, MS Platform Builder C++, or GCC).

3.9 WinDriver USB Architecture

To access your hardware, your application calls the WinDriver kernel module using functions from the WinDriver USB API. The high-level functions utilize the low-level functions, which use IOCTLs to enable communication between the WinDriver kernel module and your user-mode application. The WinDriver kernel module accesses your USB device resources through the native operating system calls.

There are two layers responsible for abstracting the USB device to the USB device driver. The upper layer is the **USB Driver (USBD)** layer, which includes the USB Hub Driver and the USB Core Driver. The lower level is the **Host Controller Driver (HCD)** layer. The division of duties between the HCD and USBD layers is not defined and is operating system dependent. Both the HCD and USBD are software interfaces and components of the operating system, where the HCD layer represents a lower level of abstraction.

The **HCD** is the software layer that provides an abstraction of the host controller hardware, while the **USBD** provides an abstraction of the USB device and the data transfer between the host software and the function of the USB device.

The **USBD** communicates with its clients (the specific device driver, for example) through the USB Driver Interface (**USBDI**). At the lower level, the Core Driver and USB Hub Driver implement the hardware access and data transfer by communicating with the HCD using the Host Controller Driver Interface (**HCDI**).

The USB Hub Driver is responsible for identifying the addition and removal of devices from a particular hub. When the Hub Driver receives a signal that a device was attached or detached, it uses additional host software and the USB Core Driver to recognize and configure the device. The software implementing the configuration can include the hub driver, the device driver, and other software.

WinDriver USB abstracts the configuration procedure and hardware access described above for the developer. With WinDriver's USB API, developers can perform all the hardware-related operations without having to master the lower-level implementation for supporting these operations.

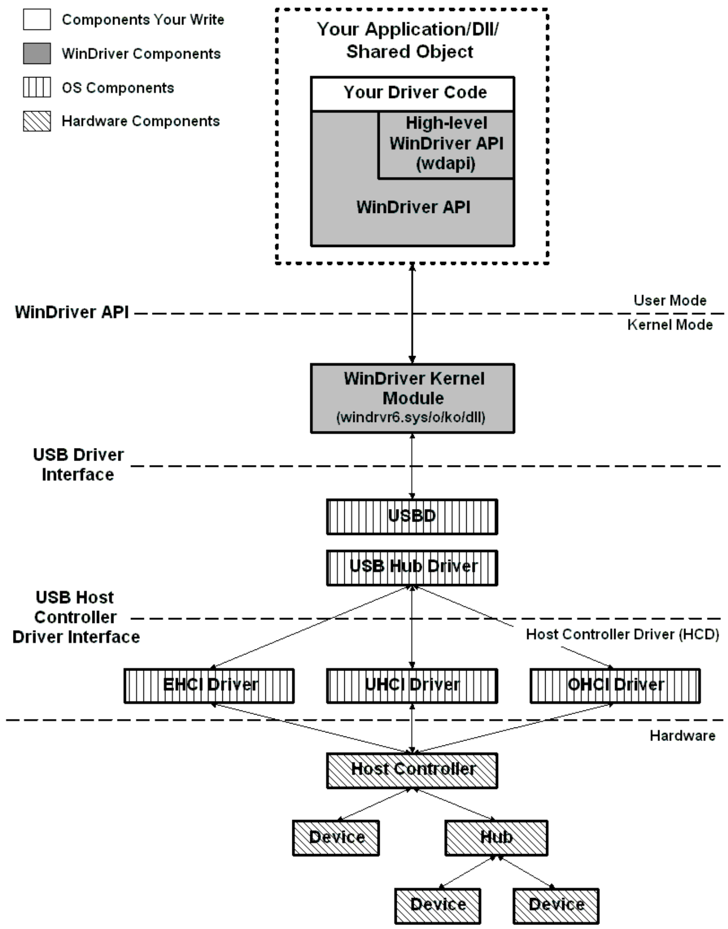


Figure 3.4: WinDriver USB Architecture

3.10 Which Drivers Can I Write with WinDriver USB?

Almost all monolithic drivers (drivers that need to access specific USB devices) can be written with WinDriver USB. In cases where a standard driver is required, e.g. NDIS driver, SCSI driver, Display driver, USB to Serial port drivers, USB layered drivers, etc., use KernelDriver USB (also from Jungo).

For quicker development time, select WinDriver USB over KernelDriver USB whenever possible.

Chapter 4

Installing WinDriver

This chapter takes you through the WinDriver installation process, and shows you how to verify that your WinDriver is properly installed. The last section discusses the uninstall procedure.

4.1 System Requirements

4.1.1 For Windows 98/Me

- Any x86 32-bit processor.
- Any 32-bit development environment supporting C, VB or Delphi.

4.1.2 For Windows 2000/XP/Server 2003

- Any x86 32-bit or 64-bit (x64: AMD64 or Intel EM64T) processor.
- Any development environment supporting C, .NET, VB or Delphi.

4.1.3 For Windows CE / Windows Mobile 5.0

- An x86 / MIPS / ARM Windows CE 4.x – 5.0 (.NET) target platform
  or:
  ARMV4I Windows Mobile 5.0 target platform.
- Windows 2000/XP/Server 2003 host development platform.
- For **Windows CE 4.x – 5.0**: Microsoft eMbedded Visual C++ with a corresponding target SDK **OR** Microsoft Platform Builder with a corresponding BSP (Board Support Package) for the target platform.

  For **Windows Mobile 5.0**: Microsoft Visual Studio (MSDEV) .NET 2005.

4.1.4 For Linux

- Any 32-bit x86 processor with a Linux 2.4.x or 2.6.x kernel
  or:
  Any 64-bit x86 AMD64 or Intel EM64T (**x86_64**) or Itanium and Itanium 2 (**IA64**) processor – with a Linux 2.4.x or 2.6.x kernel
  or:
  Any PowerPC 32-bit processor with a Linux 2.4.x or 2.6.x kernel.
- A GCC compiler.

  NOTE
  The version of the GCC compiler should match the compiler version used for building the running Linux kernel.

- Any 32-bit or 64-bit development environment (depending on your target configuration) supporting C for user mode.
- On your development PC: **glibc2.3.x**.
- **libstdc++.so.5** is required for running GUI WinDriver applications (e.g. DriverWizard [5] ; Debug Monitor [7.2]).

4.2 WinDriver Installation Process

The WinDriver CD contains all versions of WinDriver for all the different operating systems. The CD's root directory contains the Windows 98/Me/2000/XP/Server 2003 and Windows CE version. This will automatically begin when you insert the CD into your CD drive. The other versions of WinDriver are located in sub-directories, i.e. **Linux/**, **Wince/**, etc.

4.2.1 Windows 98/Me/2000/XP/Server 2003 WinDriver Installation Instructions

NOTE
You must have administrative privileges in order to install WinDriver on Windows 98/Me/2000/XP/Server 2003.

1. Insert the WinDriver CD into your CD-ROM drive.
   When installing WinDriver by downloading it from Jungo's web site instead of using the WinDriver CD, double click the downloaded installation file – **WD802.EXE** – and go to step 3.
2. Wait a few seconds until the installation program starts automatically. If for some reason it does not start automatically, double-click the file **WD802.EXE** and click the **Install WinDriver** button.
3. Read the license agreement carefully, and click **Yes** if you accept its terms.
4. Choose the destination location in which to install WinDriver.
5. In the **Setup Type** screen, choose one of the following:
   - **Typical** – install all WinDriver modules (generic WinDriver toolkit + specific chipset APIs).
   - **Compact –** install only the generic WinDriver toolkit.
   - **Custom** – select which WinDriver modules to install.
6. After the installer finishes copying the required files, choose whether to view the Quick Start guides.
7. You may be prompted to reboot your computer.

NOTE
The WinDriver installation defines a **WD_BASEDIR** environment variable, which is set to point to the location of your WinDriver directory, as selected during the installation. This variable is used during the DriverWizard [5] code generation – it determines the default directory for saving your generated code and is used in the include paths of the generated project/make files.

Therefore, if you decide to change the name and/or location of your WinDriver directory after the installation, you should also edit the value of the WD_BASEDIR environment variable and set it to point to the location of your new WinDriver directory. You can edit the value of WD_BASEDIR by following these steps:

1. Open the **System Properties** dialogue: **Start | System | Control Panel | System**.
2. In the **Advanced** tab, click the **Environment Variables** button.
3. In the **System variables** box, select the WD_BASEDIR variable and click the **Edit ...** button or double-click the mouse on the variable.
4. In the **Edit System Variable** dialogue, replace the **Variable Value** with the full path to your new WinDriver directory, then click **OK**, and click **OK** again from the **System Properties** dialogue.

The following steps are for registered users only:

In order to register your copy of WinDriver with the license you received from Jungo, follow the steps below:

8. Activate DriverWizard GUI (**Start | Programs | WinDriver | DriverWizard**).
9. Select the **Register WinDriver** option from the **File** menu and insert the license string you received from Jungo. Click the **Activate License** button.
10. To register source code that you developed during the evaluation period, refer to the documentation of `WDU_Init()` [A.3.1].

4.2.2 Windows CE / Windows Mobile 5.0 WinDriver Installation Instructions

4.2.2.1 Installing WinDriver CE when Building New CE-Based Platforms

The following instructions apply to platform developers who build Windows CE kernel images using Windows CE Platform Builder.

> **NOTE**
> We recommend that you read Microsoft's documentation and understand the Windows CE and device driver integration procedure before you perform the installation.

1. Edit the project registry file to match your target hardware. If you select to use the WinDriver component, as outlined in step 2, the registry file to modify is **WinDriver\samples\wince_install \<TARGET_CPU>\WinDriver.reg** (e.g. **WinDriver\samples\wince_install\ARMV4I\ WinDriver.reg**). Otherwise, modify the **WinDriver\samples\wince_install\project_wd.reg** file.

2. You can simplify the driver integration into your Windows CE platform by following the procedure described in this step before the Sysgen platform compilation stage.

   NOTE:

   - This procedure provides a convenience method for integrating WinDriver into your Windows CE platform. If you select not to use this method, you will need to perform the manual integration steps described in step 4 below after the Sysgen stage.
   - The procedure described in this step also adds the WinDriver kernel module (**windrvr6.dll**) to your OS image. This is a necessary step if you want the WinDriver CE kernel file (**windrvr6.dll**) to be a permanent part of the Windows CE image (**NK.BIN**), which is the case if you select to transfer the file to your target platform using a floppy disk. However, if you prefer to have the file **windrvr6.dll** loaded on demand via the CESH/PPSH services, you need to perform the manual integration method described in step 4 instead of performing the procedure described in the present step.

   (a) Run Microsoft **Platform Builder** and open your platform.

   (b) From the **File** menu select **Manage Catalog Items....** and then click the **Import...** button and select the **WinDriver.cec** file from the relevant **WinDriver\samples\wince_install\<TARGET_CPU>** directory (e.g. **WinDriver\samples\wince_install\ARMV4I**).
   This will add a WinDriver component to the Platform Builder Catalog.

(c) In the **Catalog** view, right-click the mouse on the **WinDriver Component** node in the **Third Party** tree and select **Add to OS design**.

3. Compile your Windows CE platform (Sysgen stage).

4. If you have chosen not to perform the procedure described in step 2 above, perform the following steps after the Sysgen stage in order to manually integrate the driver into your platform.

   NOTE: If you followed the procedure described in step 2, skip this step and go directly to step 5.

   (a) Run Microsoft **Platform Builder** and open your platform.

   (b) Select **Open Build Release Directory** from the **Build** menu.

   (c) Copy the WinDriver CE kernel file – **WinDriver\redist\<TARGET_CPU>\windrvr6.dll** – to the **%_FLATRELEASEDIR%** sub-directory on the target development platform (should be the current directory in the new command window).

   (d) Append the contents of the **project_wd.reg** file in the **WinDriver\samples\wince_install** directory to the **project.reg** file in the **%_FLATRELEASEDIR%** sub-directory.

   (e) Append the contents of the **project_wd.bib** file in the **WinDriver\samples\wince_install** directory to the **project.bib** file in the **%_FLATRELEASEDIR%** sub-directory.

   This step is only necessary if you want the WinDriver CE kernel file (**windrvr6.dll**) to be a permanent part of the Windows CE image (**NK.BIN**), which is the case if you select to transfer the file to your target platform using a floppy disk. If you prefer to have the file **windrvr6.dll** loaded on demand via the CESH/PPSH services, you do not need to carry out this step until you build a permanent kernel.

5. Select **Make Image** from the **Build** menu and name the new image **NK.BIN**.

6. Download your new kernel to the target platform and initialize it either by selecting **Download/Initialize** from the **Target** menu or by using a floppy disk.

7. Restart your target CE platform. The WinDriver CE kernel will automatically load.

8. Compile and run the sample programs to make sure that WinDriver CE is loaded and is functioning correctly (see section 4.4.2, which describes how to check your installation).

4.2.2.2 Installing WinDriver CE when Developing Applications for Windows CE / Windows Mobile 5.0 Computers

NOTE
Unless otherwise specified, "Windows CE" references in this section include Windows CE 4.x – 5.x and Windows Mobile 5.0.

The following instructions apply to driver developers who do not build the Windows CE kernel, but only download their drivers, built using Microsoft eMbedded Visual C++ (Windows CE 4.x – 5.x) or MSDEV .NET 2005 (Windows Mobile 5.0), to a ready-made Windows CE platform:

1. Insert the WinDriver CD into your Windows host CD drive.
2. Exit the automatic installation.
3. Double click the **CD_SETUP.EXE** file found in the **WINCE**\ directory on the CD. This will copy all required WinDriver files to your host development platform.
4. Copy WinDriver's kernel module – **windrvr6.dll** – from the **WinDriver****redist****WINCE****<TARGET_CPU>** directory on the Windows host development PC to the **Windows**\ directory on your target Windows CE platform.
5. Add WinDriver to the list of device drivers Windows CE loads on boot:
   - Modify the registry according to the entries documented in the file **WinDriver****samples****wince_install**\ **project_wd.reg**. This can be done using the Windows CE Pocket Registry Editor on the hand-held CE computer or by using the Remote CE Registry Editor Tool supplied with MS eMbedded Visual C++ (Windows CE 4.x – 5.x) / MSDEV .NET 2005 (Windows Mobile 5.0). Note that in order to use the Remote CE Registry Editor tool you will need to have Windows CE Services installed on your Windows host platform.
   - On Windows Mobile 5.0 the operating system's security scheme prevents the loading of unsigned drivers at boot time, therefore the WinDriver kernel module has to be reloaded after boot. To load WinDriver on the target Windows Mobile 5.0 platform every time the OS is started, copy the **WinDriver****redist****Windows_Mobile_5_ARMV4I**\ **wdreg.exe** utility to the **Windows****StartUp**\ directory on the target.
6. Restart your target CE computer. The WinDriver CE kernel will automatically load. You will have to do a warm reset rather than just suspend/resume (use the reset or power button on your target CE computer).

7. Compile and run the sample programs to make sure that WinDriver CE is loaded and is functioning correctly (see section 4.4, which describes how to check your installation).

4.2.2.3 Windows CE Installation Note

The WinDriver installation on the host Windows 2000/XP/Server 2003 PC defines a **WD_BASEDIR** environment variable, which is set to point to the location of your WinDriver directory, as selected during the installation. This variable is used during the DriverWizard [5] code generation – it determines the default directory for saving your generated code and is used in the include paths of the generated project/make files.

Therefore, if you decide to change the name and/or location of your host WinDriver directory after the installation, you should also edit the value of the WD_BASEDIR environment variable and set it to point to the location of your new WinDriver directory. You can edit the value of WD_BASEDIR by following these steps:

1. Open the **System Properties** dialogue: **Start | System | Control Panel | System**.
2. In the **Advanced** tab, click the **Environment Variables** button.
3. In the **System variables** box, select the WD_BASEDIR variable and click the **Edit ...** button or double-click the mouse on the variable.
4. In the **Edit System Variable** dialogue, replace the **Variable Value** with the full path to your new WinDriver directory, then click **OK**, and click **OK** again from the **System Properties** dialogue.

Note that if you install the WinDriver Windows 98/Me/2000/XP/Server 2003 tool-kit on the same host PC, the installation will override the value of the WD_BASEDIR variable from the Windows CE installation.

4.2.3 Linux WinDriver Installation Instructions

4.2.3.1 Preparing the System for Installation

In Linux, kernel modules must be compiled with the same header files that the kernel itself was compiled with. Since WinDriver installs the kernel module **windrvr6.o/.ko**, it must compile with the header files of the Linux kernel during the installation process.

Therefore, before you install WinDriver for Linux, verify that the Linux source code and the file **versions.h** are installed on your machine:

Install the Linux kernel source code:

- If you have yet to install Linux, install it, including the kernel source code, by following the instructions for your Linux distribution.
- If Linux is already installed on your machine, check whether the Linux source code was installed. You can do this by looking for 'linux' in the **/usr/src** directory. If the source code is not installed, either install it, or reinstall Linux with the source code, by following the instructions for your Linux distribution.

Install version.h:

- The file **version.h** is created when you first compile the Linux kernel source code. Some distributions provide a compiled kernel without the file **version.h**. Look under **/usr/src/linux/include/linux** to see if you have this file. If you do not, follow these steps in order to install the file:
  1. Type:
     `$ make xconfig`
  2. Save the configuration by choosing **Save and Exit**.
  3. Type:
     `$ make dep`

In order to run GUI WinDriver applications (e.g. DriverWizard [5] ; Debug Monitor [7.2]) you must also have version 5.0 of the **libstdc++** library – **libstdc++.so.5**. If you do not have this file, install it from the relevant RPM in your Linux distribution (e.g. **compat-libstdc++**).

Before proceeding with the installation, you must also make sure that you have a 'linux' symbolic link. If you do not, create one by typing:
`/usr/src$ ln -s <target kernel>/ linux`
For example, for the Linux 2.4 kernel type:
`/usr/src$ ln -s linux-2.4/ linux`

4.2.3.2 Installation

1. Insert the WinDriver CD into your Linux machine's CD drive or copy the downloaded file to your preferred directory.
2. Change directory to your preferred installation directory, for example to your home directory:
   `$ cd ~`
3. Extract the WinDriver distribution file – **WD802LN.tgz**:
   `$ tar xvzf /<file location>/WD802LN.tgz`

   For example:

   - From a CD:
     `$ tar xvzf /mnt/cdrom/LINUX/WD802LN.tgz`
   - From a downloaded file:
     `$ tar xvzf /home/username/WD802LN.tgz`
4. Change directory to your WinDriver **redist/** directory (the tar automatically creates a **WinDriver/** directory):
   `$ cd <WinDriver directory path>/redist`
5. Install WinDriver:
   (a) `<WinDriver directory>/redist$ ./configure`

   > **NOTE**
   > The **configure** script creates a **makefile** based on your specific running kernel. You may run the **configure** script based on another kernel source you have installed, by adding the flag **--with-kernel-source=<path>** to the configure script. The <path> is the full path to the kernel source directory, e.g. **/usr/src/linux**.

   (b) `<WinDriver directory>/redist$ make`

   (c) Become super user:
   `<WinDriver directory>/redist$ su`

   (d) Install the driver:
   `<WinDriver directory>/redist# make install`
6. Create a symbolic link so that you can easily launch the DriverWizard GUI:
   `$ ln -s <full path to WinDriver>/wizard/wdwizard/ usr/bin/wdwizard`
7. Change the read and execute permissions on the file **wdwizard** so that ordinary users can access this program.

8. Change the user and group IDs and give read/write permissions to the device file **/dev/windrvr6** depending on how you wish to allow users to access hardware through the device.

   If you are using a Linux 2.6.x kernel that has the **udev** file system, change the permissions by modifying your **/etc/udev/permissions.d/50-udev.permissions** file. For example, add the following line to provide read and write permissions: `windrvr6:root:root:0666`

   Otherwise, use the **`chmod`** command, for example:
   `chmod /dev/windrvr6 666`

9. Define a new **`WD_BASEDIR`** environment variable and set it to point to the location of your WinDriver directory, as selected during the installation. This variable is used in the make and source files of the WinDriver samples and generated DriverWizard [5] code and is also used to determine the default directory for saving your generated DriverWizard project. If you do not define this variable you will be instructed to do so when attempting to build the sample/generated code using the WinDriver makefiles.
   NOTE: If you decide to change the name and/or location of your WinDriver directory after the installation, you should also edit the value of the `WD_BASEDIR` environment variable and set it to point to the location of your new WinDriver directory.

10. You can now start using WinDriver to access your hardware and generate your driver code!

> **TIP**
> You can use the **wdreg** script to load the WinDriver kernel module [10.3].
> To automatically load **windrvr6.o/.ko** on each boot, run the **wdreg** script from the target Linux **/etc/rc.d/rc.local** file:
> **`wdreg windrvr6`**

The following steps are for registered users only

In order to register your copy of WinDriver with the license you received from Jungo, follow the steps below:

11. Activate the DriverWizard GUI:
    `<path to WinDriver>/wizard/wdwizard`

12. Select the **Register WinDriver** option from the **File** menu and insert the license string you received from Jungo.

13. Click the **Activate License** button.

14. To register source code that you developed during the evaluation period, refer to the documentation of `WDU_Init()` [A.3.1].

4.2.3.3 Restricting Hardware Access on Linux

CAUTION!
Since **/dev/windrvr6** gives direct hardware access to user programs, it may compromise kernel stability on multi-user Linux systems. Please restrict access to the DriverWizard and the device file **/dev/windrvr6** to trusted users.

For security reasons the WinDriver installation script does not automatically perform the steps of changing the permissions on **/dev/windrvr6** and the DriverWizard executable (**wdwizard**).

4.3 Upgrading Your Installation

To upgrade to a new version of WinDriver on Windows, follow the steps outlined in section 4.2.1, which illustrate the process of installing WinDriver for Windows 98/Me/2000/XP/Server 2003. You can either choose to overwrite the existing installation or install to a separate directory.

After installation, start DriverWizard and enter the new license string, if you have received one. This completes the upgrade of WinDriver.

To upgrade your source code, pass the new license string as a parameter to `WDU_Init()` [A.3.1] (or to `WD_License()`, when using the old `WD_UsbXXX()` APIs).

The procedure for upgrading your installation on other operating systems is the same as the one described above. Please check the respective installation sections for installation details.

4.4 Checking Your Installation

4.4.1 On Your Windows and Linux Machines

1. Start DriverWizard:
   On Windows, by choosing **Programs | WinDriver | DriverWizard** from the **Start** menu, or using the shortcut that is automatically created on your Desktop. A third option for activating the DriverWizard on Windows is by running **wdwizard.exe** from a command prompt under the **wizard** sub-directory.
   On Linux you can access the wizard application via the file manager under the **wizard** sub-directory, or run the wizard application via a shell.

2. Make sure that your WinDriver license is installed (see section 4.2, which explains how to install WinDriver). If you are an evaluation version user, you do not need to install a license.

4.4.2 On Your Windows CE Machine

1. Start DriverWizard on your Windows host machine by choosing **Programs | WinDriver | DriverWizard** from the **Start Menu**.

2. Make sure that your WinDriver license is installed. If you are an evaluation version user, you do not need to install a license.

3. Plug your device into the computer, and verify that DriverWizard detects it.

4. Activate Visual C++ for CE.

5. Load one of the WinDriver samples, e.g., **WinDriver\samples\speaker\speaker.dsw**.

6. Set the target platform to x86em in the Visual C++ WCE configuration toolbar.

7. Compile and run the speaker sample. The Windows host machine's speaker should be activated from within the CE emulation environment.

4.5 Uninstalling WinDriver

This section will help you to uninstall either the evaluation or registered version of WinDriver.

4.5.1 On Windows 98/Me/2000/XP/Server 2003

NOTES

- For **Windows 98/Me**, replace references to **wdreg** below with **wdreg16**.
- For **Windows 2000/XP/Server 2003**, you can also use the **wdreg_gui.exe** utility instead of **wdreg.exe**.
- **wdreg.exe**, **wdreg_gui.exe** and **wdreg16.exe** are found under the **WinDriver\util** directory (see Chapter 10 for details regarding these utilities).

1. Close any open WinDriver applications, including DriverWizard, the Debug Monitor (**wddebug_gui.exe**) and user-specific applications.
2. On Plug-and-Play Windows systems (**Windows 98 / Me / 2000 / XP / Server 2003**): Uninstall all Plug-and-Play devices (USB/PCI/PCMCIA) that have been registered with WinDriver via an INF file:
   - On **Windows 2000/XP/Server 2003**: Uninstall the device using the **wdreg** utility:
     ```
     wdreg -inf <path to the INF file> uninstall
     ```
     On **Windows 98/Me**: Uninstall (Remove) the device manually from the Device Manager.
   - Verify that no INF files that register your device(s) with WinDriver's kernel module (**windrvr6.sys**) are found in the **%windir%\inf** directory and/or **%windir%\inf\other** directory (Windows 98/Me).
3. Uninstall WinDriver:
   - **On the development PC**, on which you installed the WinDriver toolkit: Run **Start | WinDriver | Uninstall**, **OR** run the **uninstall.exe** utility from the **WinDriver** installation directory.

     The uninstall will stop and unload the WinDriver kernel module (**windrvr6.sys**); delete the copy of the **windrvr6.inf** file from the **%windir%\inf** directory (on Windows 2000/XP/Server 2003) or **%windir%\inf \other** directory (on Windows 98/Me); delete WinDriver from Windows' **Start** menu; delete the **WinDriver** installation directory

(except for files that you added to this directory); and delete the short-cut icons to the DriverWizard and Debug Monitor utilities from the Desktop.

- **On a target PC**, on which you installed the WinDriver kernel module (**windrvr6.sys**), but not the entire WinDriver toolkit:
  Use the **wdreg** utility to stop and unload the driver:
  `wdreg -inf <path to windrvr6.inf> uninstall`

  (On the development PC, the relevant **wdreg** uninstall command is executed for you by the uninstall utility).

NOTES

- If there are open handles to WinDriver when attempting to uninstall it (either using the **uninstall** utility or by running the **wdreg** uninstall command directly) – for example if there is an open WinDriver application or a connected Plug-and-Play device that has been registered to work with WinDriver via an INF file (on Windows 98/Me/2000/XP/Server 2003) – an appropriate warning message will be displayed. The message will provide you with the option to either close the open application(s) / uninstall/disconnect the relevant device(s), and **Retry** to uninstall the driver; or **Cancel** the uninstall of the driver, in which case the **windrvr6.sys** kernel driver will not be uninstalled. This ensures that you do not uninstall the WinDriver kernel module (**windrvr6.sys**) as long as it is being used.
- You can check if the WinDriver kernel module is loaded by running the Debug Monitor utility (**WinDriver\util\wddebug_gui.exe**). When the driver is loaded the Debug Monitor log displays driver and OS information; otherwise it displays a relevant error message.
  On the development PC the uninstall command will delete this utility, therefore in order to use it after you execute the uninstallation, create a copy of **wddebug_gui.exe** before performing the uninstall procedure.

4. If **windrvr6.sys** was successfully unloaded, erase the following files (if they exist):
   - **%windir%\system32\drivers\windrvr6.sys**
   - **%windir%\inf\windrvr6.inf** (Windows 2000/XP/Server 2003)
   - **%windir%\inf\Jungowindrvr6.inf** (Windows 98/Me)
   - **%windir%\system32\wdapi802.dll**
   - **%windir%\sysWOW64\wdapi802.dll** (Windows x64)
5. Reboot the computer.

4.5.2 On Linux

NOTE
You must be logged in as root to perform the uninstall procedure.

1. Verify that the WinDriver module is not being used by another program:
   - View a list of modules and the programs using each of them:
     `/# ` **`/sbin/lsmod`**
   - Close any applications that are using the WinDriver module.
   - Unload any modules that are using the WinDriver module:
     `/sbin# ` **`rmmod`**
2. Unload the WinDriver module:
   `/sbin# ` **`rmmod windrvr6`**
3. If you are not using a Linux 2.6.x kernel that supports the **udev** file system, remove the old device node in the **/dev** directory:
   `/# ` **`rm -rf /dev/windrvr6`**
4. Remove the file **.windriver.rc** from the **/etc** directory:
   `/# ` **`rm -rf /etc/.windriver.rc`**
5. Remove the file **.windriver.rc** from **$HOME**:
   `/# ` **`rm -rf $HOME/.windriver.rc`**
6. If you created a symbolic link to DriverWizard, delete the link using the command:
   `/# ` **`rm -f /usr/bin/wdwizard`**
7. Delete the WinDriver installation directory using the command:
   `/# ` **`rm -rf ~/WinDriver`**
8. Erase the following shared object file, if it exists:
   /usr/lib/libwdapi802.so (32-bit PowerPC, 32-bit x86 or 64-bit IA64) /
   /usr/lib64/libwdapi802.so (64-bit x86).

Chapter 5

Using DriverWizard

This chapter describes WinDriver DriverWizard's hardware diagnostics and driver code generation capabilities.
To find out how you can use the WinDriver USB Device DriverWizard to develop device firmware, refer to Chapter 12.

5.1 An Overview

DriverWizard (included in the WinDriver toolkit) is a GUI-based diagnostics and driver generation tool that allows you to write to and read from the hardware, before writing a single line of code. The hardware is diagnosed through a Graphical User Interface – the device's configuration and pipes information is displayed, data can be transferred on the pipes, the pipes can be reset, etc. Once the device is operating to your satisfaction, DriverWizard creates the skeletal driver source code, with functions to access your hardware's resources.

If you are developing a driver for a device that is based on one of the enhanced-support USB chipsets (The Cypress EZ-USB family, Microchip PIC18F4550, Philips PDIUSBD12, Texas Instruments TUSB3410, TUSB3210, TUSB2136, TUSB5052, Silicon Laboratories C8051F320), we recommend you read Chapter 8, which explains WinDriver's enhanced support for specific chipsets, before starting your driver development.

DriverWizard can be used to diagnose your hardware and can generate an INF file for hardware running under Windows 98/Me/2000/XP/Server 2003. Avoid using DriverWizard to generate code for a device based on one of the supported USB chipsets [8], as DriverWizard generates generic code which will have to be modified

according to the specific functionality of the device in question. Preferably, use the complete source code libraries and sample applications (supplied in the package) tailored to the various USB chipsets.

DriverWizard is an excellent tool for two major phases in your HW/Driver development:

Hardware diagnostics: After the hardware has been built, attach your device to a USB port on your machine, and use DriverWizard to verify that the hardware is performing as expected.

Code generation: Once you are ready to build your code, let DriverWizard generate your driver code for you.

The code generated by DriverWizard is composed of the following elements:

Library functions for accessing each element of your device's resources (memory ranges, I/O ranges, registers and interrupts).

A 32-bit diagnostics program in console mode with which you can diagnose your device. This application utilizes the special library functions described above. Use this diagnostics program as your skeletal device driver.

A project workspace/solution that you can use to automatically load all of the project information and files into your development environment.
For Linux, DriverWizard generates the required makefile.

5.2 DriverWizard Walkthrough

To use DriverWizard:

1. **Attach your hardware to the computer:**
   Attach your device to a USB port on your computer.
2. **Run DriverWizard and select your device:**
   (a) Click **Start | Programs | WinDriver | DriverWizard** or double click the DriverWizard icon on your desktop (on Windows), or run the **wdwizard** utility from the **WinDriver/wizard/** directory.
   (b) Click **Next** in the **Choose Your Project** dialogue box.
   (c) Select your **Device** from the list of devices detected by DriverWizard.

> **NOTE**
> On Windows 98, if you do not see your USB device in the list, reconnect it and make sure the **New Hardware Found**/**Add New Hardware** wizard appears for your device. Do not close the dialogue box until you have generated an INF for your device using the steps below.

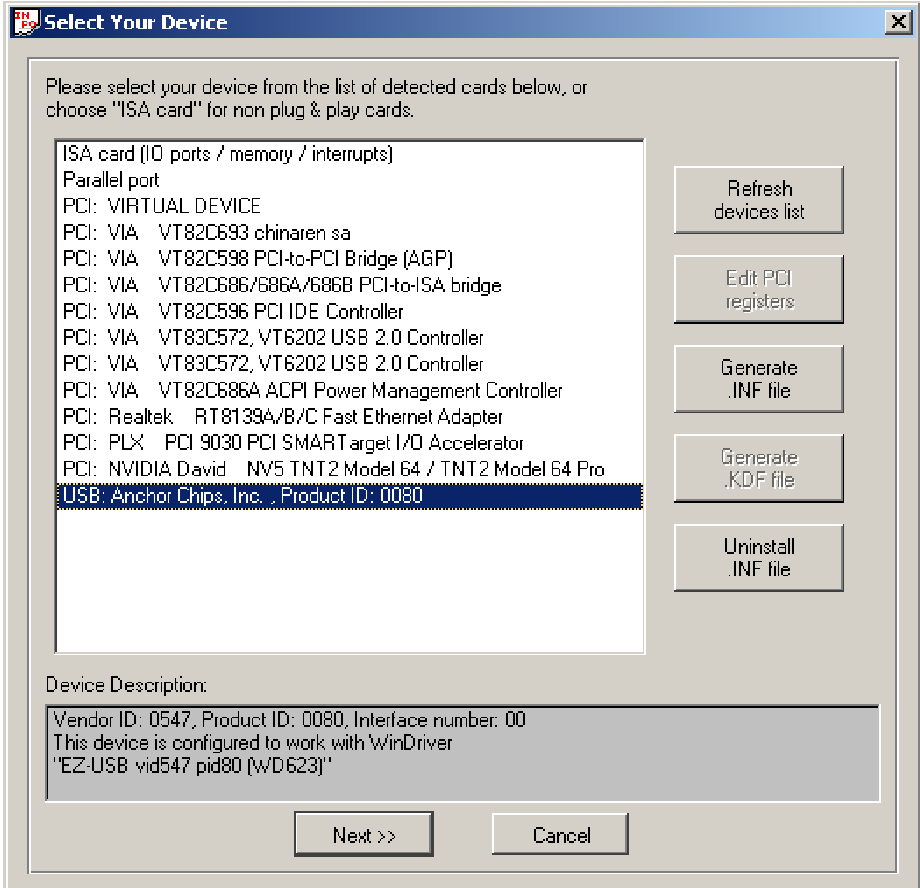


Figure 5.1: Select Your Device

3. **Generate an INF file for DriverWizard:**
Whenever developing a driver for a Plug and Play Windows operating system (i.e., Windows 98/Me/2000/XP/Server 2003) you are required to install an INF file for your device. This file will register your Plug and Play device to work with the **windrvr6.sys** driver. The file generated by the DriverWizard in this step should later be distributed to your customers using Windows 98/Me/2000/XP/Server 2003, and installed on their PCs.
The INF file you generate here is also designed to enable DriverWizard to diagnose your device. As explained earlier, this is required only when using WinDriver to support a Plug and Play device (such as USB) on a Plug and

Play system (Windows 98/Me/2000/XP/Server 2003). Additional information concerning the need for an INF file is explained in section 11.3.1.

If you do not need to generate an INF file (e.g. if you are using DriverWizard on Linux), skip this step and proceed to the next one.

To generate the INF file with the DriverWizard, follow the steps below:

(a) In the **Select Your Device** screen, click the **Generate .INF file** button or click **Next**.

(b) DriverWizard will display information detected for your device – Vendor ID, Product ID, Device Class, manufacturer name and device name – and allow you to modify this information.

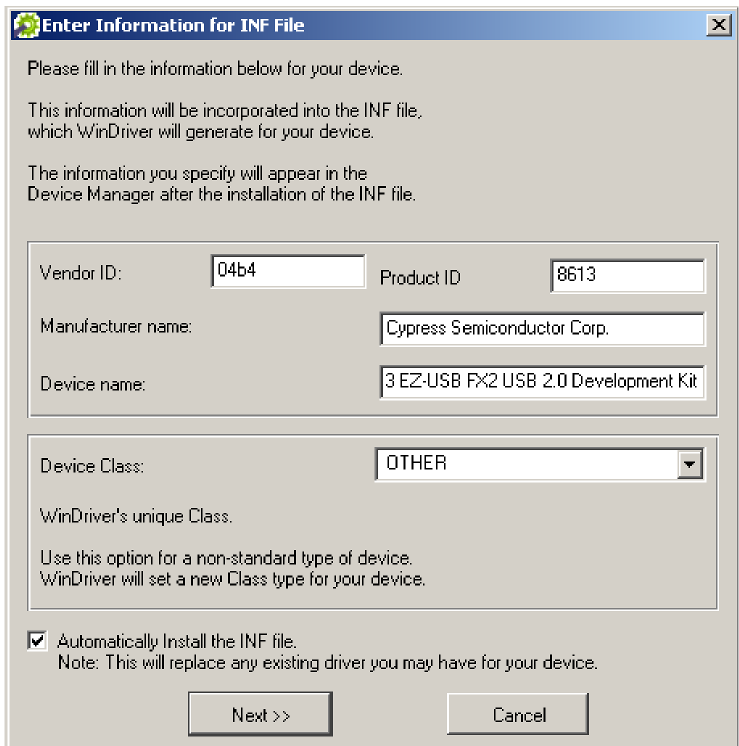


Figure 5.2: DriverWizard INF File Information

(c) For multiple-interface USB devices, you can select to generate an INF file either for the composite device or for a specific interface.

- When selecting to generate an INF file for a specific interface of a multi-interface USB device the INF information dialogue will indicate for which interface the INF file is generated.

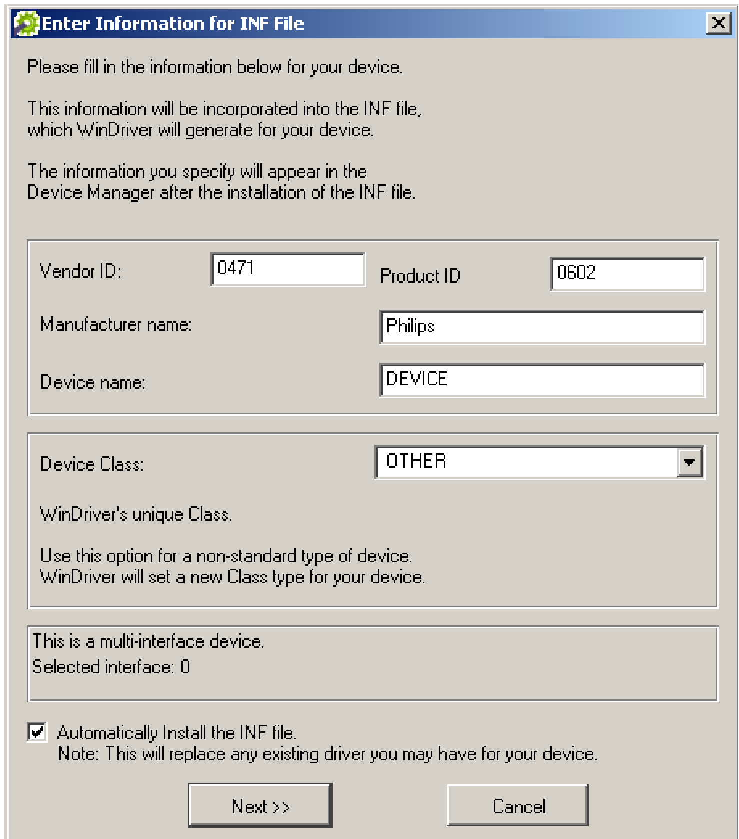


Figure 5.3: DriverWizard Multi-Interface INF File Information – Specific Interface

- When selecting to generate an INF file for a composite device of a multi-interface USB device, the INF information dialogue provides you with the option to either generate an INF file for the root device itself, or generate an INF file for specific interfaces, which you can select from the dialogue.
  Selecting to generate an INF file for the root device will enable you to handle multiple active interfaces simultaneously.

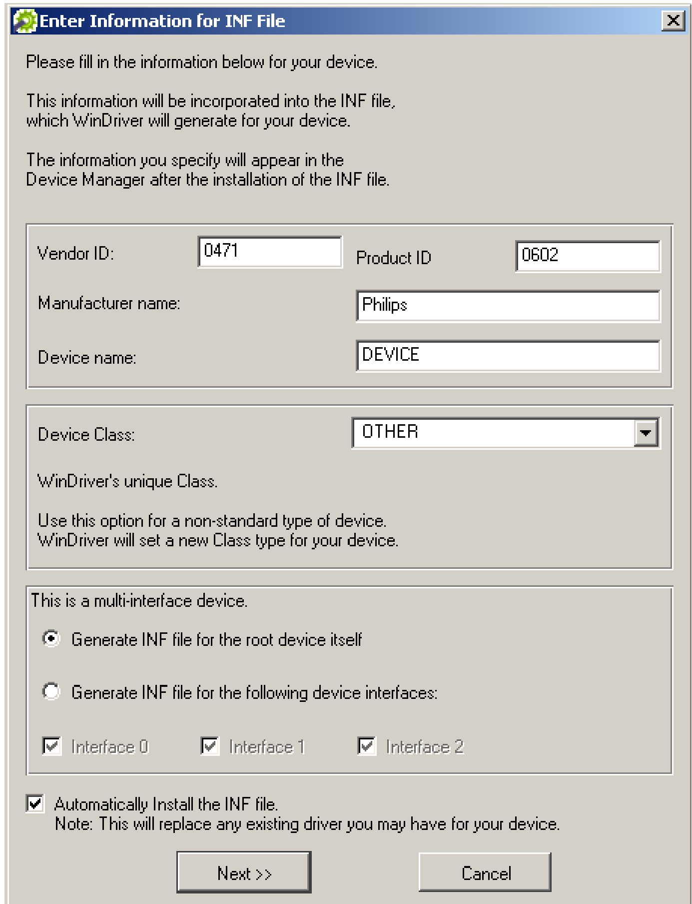


Figure 5.4: DriverWizard Multi-Interface INF File Information – Composite Device

(d) When you are done, click **Next** and choose the directory in which you wish to store the generated INF file. DriverWizard will then automatically generate the INF file for you.

On **Windows 2000/XP/Server 2003** you can choose to automatically install the INF file from the DriverWizard by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation dialogue (this option is checked by default for USB devices). On **Windows 98/Me** you must install the INF file manually, using Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as explained in section 11.3.

If the automatic INF file installation on Windows 2000/XP/Server 2003 fails, DriverWizard will notify you and provide manual installation instructions for this OS as well.

(e) When the INF file installation completes, select and open your device from the list in the **Select Your Device** screen.

4. **Uninstall the INF file of your device:**
   You can use the **Uninstall** option to uninstall the INF file of your device. Once you uninstall the INF file, the device will no longer be registered to work with the **windrvr6.sys**, and the INF file will be deleted from the Windows root directory. **If you do not need to uninstall an INF file, skip this step and proceed to the next one**.

   (a) In the **Select Your Device** screen, click the **Uninstall .INF file** button.

   (b) Select the INF file to be removed.

5. **Select the desired alternate setting:**
   The DriverWizard detects all the device's supported alternate settings and displays them. Select the desired **alternate setting** from the displayed list.

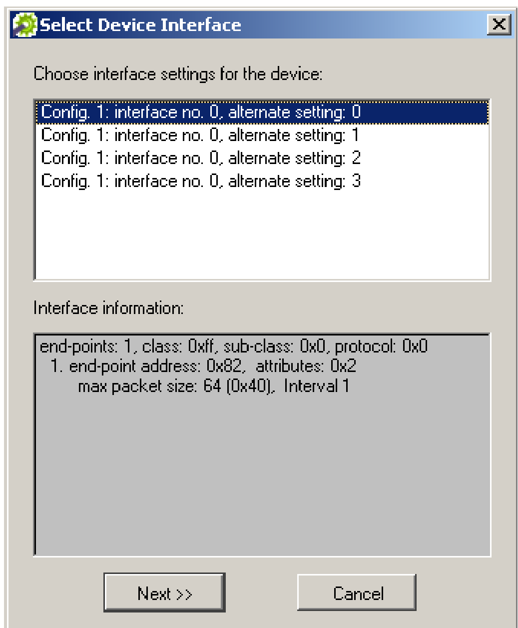


Figure 5.5: Select Device Interface

DriverWizard will display the pipes information for the selected alternate setting.

NOTE
For USB devices with only one alternate setting configured, DriverWizard automatically selects the detected alternate setting and therefore the **Select Device Interface** dialogue will not be displayed.

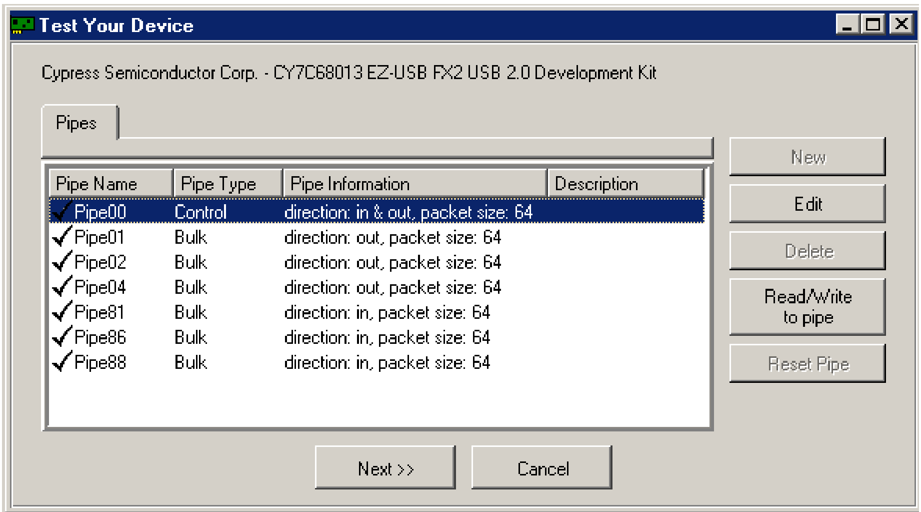


Figure 5.6: Test Your Device

6. **Diagnose your device:**
   Before writing your device driver, it is important to make sure your hardware is working as expected. Use DriverWizard to diagnose your hardware. All of your activity will be logged in the DriverWizard log so that you may later analyze your tests:

   (a) Test your USB device's pipes: DriverWizard shows the pipes detected for the selected alternate setting. To perform USB data transfers on the pipes, follow these steps:

      i. Select the desired pipe.

      ii. For a control pipe (a bidirectional pipe), click **Read/Write to Pipe**. A new dialogue will appear, allowing you to select a standard USB request or define a custom request, as demonstrated in Figure 5.7.

      When you select one of the available standard USB requests, the setup packet information for the selected request is automatically

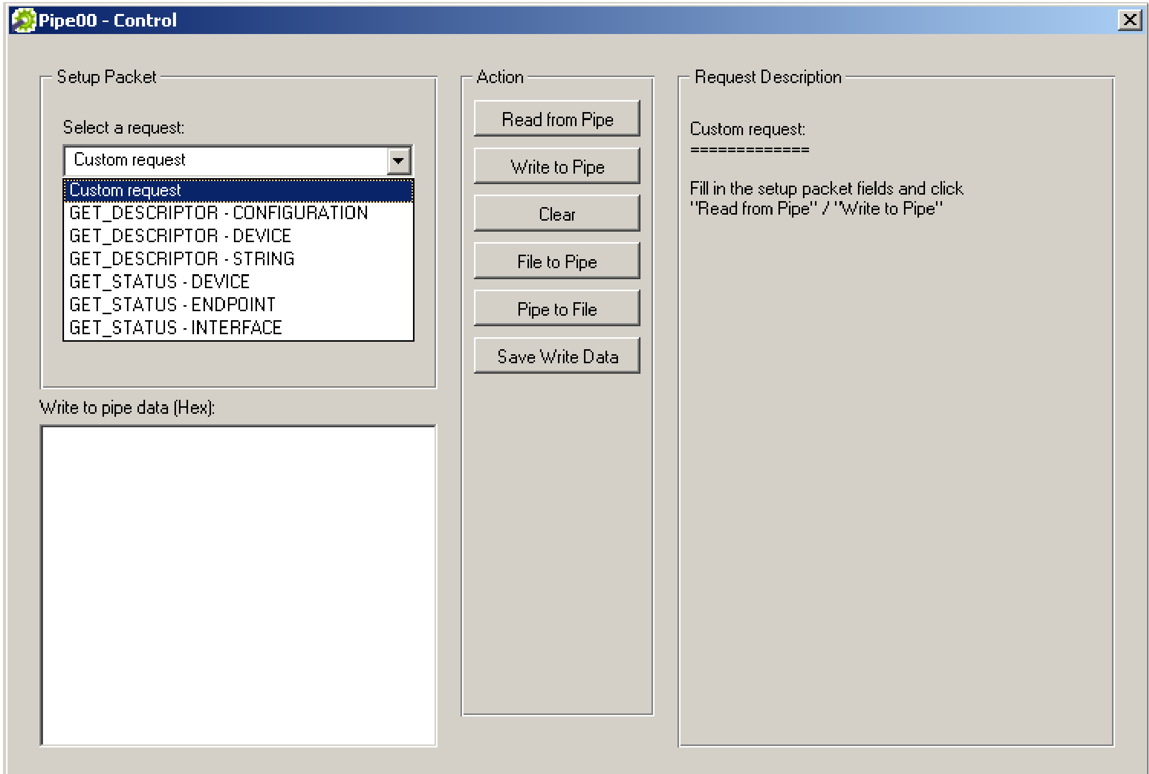


Figure 5.7: USB Requests List

filled and the request description is displayed in the **Request Description** box.

For a custom request, you are required to enter the setup packet information and write data (if exists) yourself. The size of the setup packet should be eight bytes and it should be defined using little endian byte ordering. The setup packet information should conform to the USB specification parameters (`bmRequestType`, `bRequest`, `wValue`, `wIndex`, `wLength`).

NOTE
More detailed information on the standard USB requests, on how to implement the control transfer and how to send setup packets can be found in Chapter 9.

iii. For an input pipe (moves data from device to host) click **Listen to Pipe**. To successfully accomplish this operation with devices other than HID, you need to first verify that the device sends data to the host. If no data is sent after listening for a short period of time, DriverWizard will notify you that the **Transfer Failed**.

iv. To stop reading, click **Stop Listen to Pipe**.

v. For an output pipe (moves data from host to device), click **Write to Pipe**. A new dialogue box will appear (see Figure 5.8), asking you to enter the data to write. The DriverWizard log will contain the result of the operation.

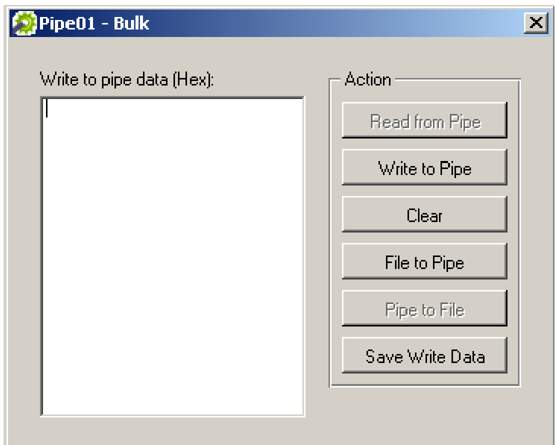


Figure 5.8: Write to Pipe

7. **Generate the skeletal driver code:**

   (a) Select **Generate Code** from the **Build** menu, or click **Next** in the **Test Your Device** dialogue window.

   (b) In the **Select Code Generation Options** dialogue box that will appear, choose the code language and development environment(s) for the generated code and select **Next** to generate the code.

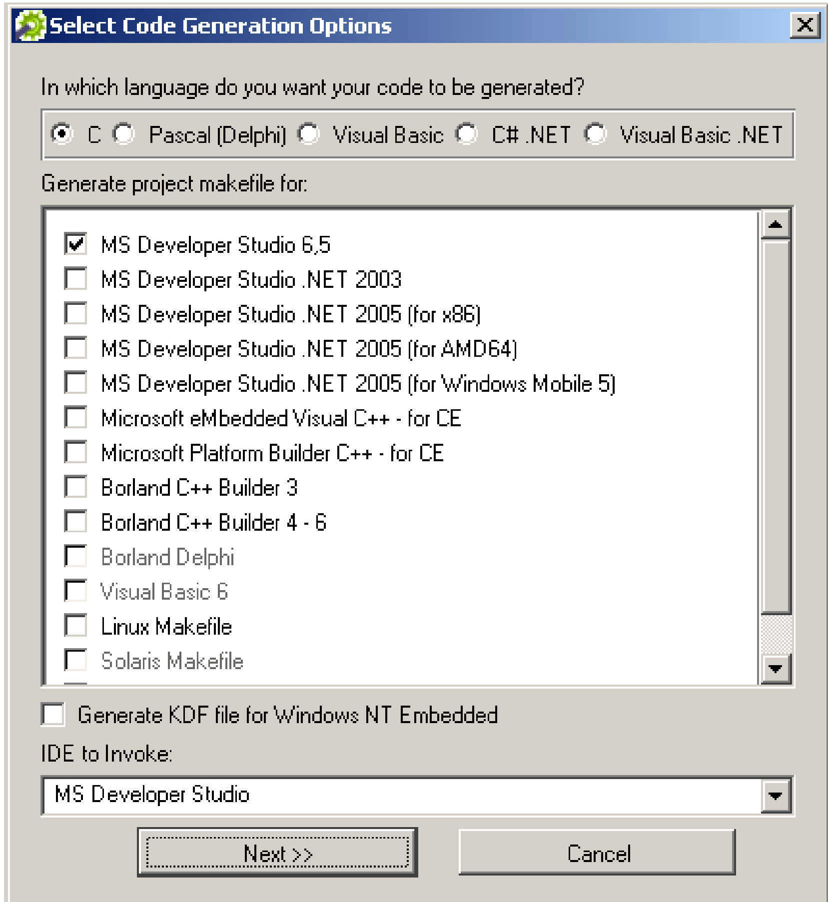


Figure 5.9: Code Generation Options

(c) Save your project (if required) and click **OK** to open your development environment with the generated driver.

(d) Close DriverWizard.

8. **Compile and run the generated code:**

   - Use this code as a starting point for your device driver. Modify where needed to perform your driver's specific functionality.

   - The source code DriverWizard creates can be compiled with any 32-bit compiler, and will run on all supported platforms (Windows 98/Me/2000/XP/Server2003, Windows CE.NET/Windows Mobile 5.0

and Linux) without modification.

5.3 DriverWizard Notes

5.3.1 Logging WinDriver API Calls

You have the option to log all the WinDriver API calls using the DriverWizard, with the API calls input and output parameters. You can select this option by selecting the **Log API calls** option from the **Tools** menu or by clicking on the **Log API calls** toolbar icon in the DriverWizard's opening window.

5.3.2 DriverWizard Logger

The wizard logger is the empty window that opens along with the **Device Resources** dialogue box when you open a new project. The logger keeps track of all of the input and output during the diagnostics stage, so that you may analyze your device's physical performance at a later time. You can save the log for future reference. When saving the project, your log is saved as well. Each log is associated with one project.

5.3.3 Automatic Code Generation

After you have finished diagnosing your device and have ensured that it runs according to your specifications, you are ready to write your driver.

5.3.3.1 Generating the Code

Choose **Generate Code** from the **Build** menu. DriverWizard will generate the source code for your driver, and place it along with the project file (**xxx.wdp**, where "xxx" is the project name). The files are saved in a directory DriverWizard creates for every development environment and operating system selected in the **Generate Code** dialogue box.

5.3.3.2 The Generated USB C Code

In the source code directory you now have a new **xxx_diag.c** source file (where **xxx** is the name you selected for your DriverWizard project). This file implements a diagnostic USB application, which demonstrates how to use WinDriver's USB API to locate and communicate with your USB device(s), including detection of Plug and Play events (device insertion/removal, etc.), performing read/write transfers on the pipes, resetting the pipes and changing the device's active alternate setting.
The generated application supports handling of multiple identical USB devices.

5.3.3.3 The Generated Visual Basic and Delphi Code

The generated DriverWizard Visual Basic and Delphi code includes similar functions and provides similar functionality as the generated C code described in section 5.3.3.2.

The generated Delphi code implements a console application (like the C code), while the Visual Basic code implements a GUI application.

5.3.3.4 The Generated C# and Visual Basic .NET Code

The generated DriverWizard C# and Visual Basic .NET code provides similar functionality as the generated C code [5.3.3.2], but from a GUI .NET program.

5.3.4 Compiling the Generated Code

5.3.4.1 Windows 98/Me/2000/XP/Server 2003 and Windows CE Compilation:

As explained above, on Windows you can select to generate project and workspace/solution files for any of the supported integrated development environments (IDEs) – MSDEV/Visual C++ 5/6, MSDEV .NET 2003/2005, Borland C++ Builder, Visual Basic 6.0, Borland Delphi, MS eMbedded Visual C++ or MS Platform Builder – and you can also select to automatically invoke your selected IDE from the wizard. You can then proceed to immediately build and run the code from your IDE.

You can also build the generated code from any other IDE that supports the selected code language and target OS. Simply create a new project file for your selected IDE, then add the generated source files to your project and compile and run the code.

NOTES

- For **Windows 98/Me/2000/XP/Server 2003**, the generated IDE files are located under an **x86**\ directory – for 32-bit projects, or **amd64**\ directory – for 64-bit projects.
- For Windows CE, note that the generated **Windows Mobile 5.0** code is targeted at the Windows Mobile 5.0 ARMV4I SDK.

5.3.4.2 Linux Compilation

Use the makefile that was created for you by DriverWizard in order to build the generated code using your favourite compiler, preferably GCC.

Chapter 6

Developing a Driver

This chapter takes you through the WinDriver driver development cycle.

> **NOTE**
> If your device is based on one of the chipsets for which WinDriver provides enhanced support (The Cypress EZ-USB family, Microchip PIC18F4550, Philips PDIUSBD12, Texas Instruments TUSB3410, TUSB3210, TUSB2136, TUSB5052, Silicon Laboratories C8051F320), read the following overview and then skip straight to Chapter 8.

6.1 Using the DriverWizard to Build a Device Driver

- Use DriverWizard to diagnose your device: View the device's configuration information, transfer data on the device's pipes, send standard requests to the control pipe and reset the pipes. Verify that your device operates as expected.
- Use DriverWizard to generate skeletal code for your device in C, C#, Visual Basic .NET, Delphi or Visual Basic. For more information about DriverWizard, refer to Chapter 5.
- If you are using one of the specific chipsets for which WinDriver offers enhanced support (The Cypress EZ-USB family, Microchip PIC18F4550, Philips PDIUSBD12, Texas Instruments TUSB3410, TUSB3210, TUSB2136, TUSB5052, Silicon Laboratories C8051F320), we recommend that you use the specific sample code provided for your chip as your skeletal driver code. For more details regarding WinDriver's enhanced support for specific chipsets, refer to Chapter 8.

- Use any C / .NET / Delphi / Visual Basic compiler (such as MSDEV/Visual C/C++, MSDEV .NET, Borland C++ Builder, Borland Delphi, Visual Basic 6.0, MS eMbedded Visual C++, MS Platform Builder C++, GCC, etc.) to compile the skeletal driver you need.
- For Linux, use any compilation environment, preferably GCC, to build your code.
- That is all you need to do in order to create your user-mode driver.

Please see Appendix A for a detailed description of WinDriver's USB API.
To learn how to implement control transfers with WinDriver, refer to Chapter 9 of the manual.

6.2 Writing the Device Driver Without the DriverWizard

There may be times when you choose to write your driver directly, without using DriverWizard. In such cases, either follow the steps outlined in this section to create a new driver project, or use one of the WinDriver samples, which most closely resembles your target driver, and modify the sample to suite your specific requirements.

6.2.1 Include the Required WinDriver Files

1. Include the relevant WinDriver header files in your driver project (all header files are found under the **WinDriver/include/** directory).
   All WinDriver projects require the **windrvr.h** header file.
   When using the `WDU_xxx` WinDriver USB API [A.1], include the **wdu_lib.h** header file (this file already includes **windrvr.h**).
   Include any other header file that provides APIs that you wish to use from your code (e.g. files from the **WinDriver/samples/shared/** directory, which provide convenient diagnostics functions.)
2. Include the relevant header files from your source code: For example, to use the USB API from the **wdu_lib.h** header file, add the following line to the code:

```
#include "wdu_lib.h"
```

3. Link your code with the **wdapi802** library/shared object:

   - For Windows 98/Me/2000/XP/Server 2003: **WinDriver\lib\<CPU>\wdapi802.lib** or **wdapi802_borland.lib** (for Borland C++ Builder), where the **<CPU>** directory is either **x86** (32-bit binaries for x86 platforms), **am64** (64-bit binaries for x64 platforms) or **am64\x86** (32-bit binaries for x64 platforms).
   - For Windows CE: **WinDriver\lib\WINCE\<CPU>\wdapi802.lib**.
   - For Linux: **WinDriver/lib/libwdapi802.so**.

   You can also include the library's source files in your project instead of linking the project with the library. The C source files are located under the **WinDriver/src/wdapi** directory.

   NOTE: When linking your project with the the **wdapi802** library/shared object, you will need to distribute the **wdapi802** DLL/shared object with your driver. For Windows, get **wdapi802.dll / wdapi802_32.dll** (for 32-bit applications targeted at 64-bit platforms) from the **WinDriver\redist** directory. For Linux, distribute **WinDriver/lib/libwdapi802.so**. For details, refer to the driver distribution instructions in Chapter 11.

4. Add any other WinDriver source files that implement API that you which to use in your code (e.g. files from the **WinDriver/samples/shared** directory.)

6.2.2 Write Your Code

1. Call `WDU_Init()` [A.3.1] at the beginning of your program to initialize WinDriver for your USB device and wait for the device-attach callback. The relevant device information will be provided in the attach callback.
2. Once the attach callback is received, you can start using one of the `WDU_Transfer()` [A.3.7] functions family to send and receive data.
3. To finish, call `WDU_Uninit()` [A.3.6] to un-register from the device.

6.3 Developing Your Driver on Windows CE Platforms

In order to register your USB device to work with WinDriver, you can perform one of two of the following:

- Call **WDU_Init()** [A.3.1] before the device is plugged into the CE system.

  OR

- You can add the following entry to the registry (can be added to your **platform.reg** file):

  ```
  [HKEY_LOCAL_MACHINE\DRIVERS\USB\LoadClients\<ID>\Default\Default\WDR]:
  "DLL"="windrvr6.dll"
  ```

  <ID> is comprised of your vendor ID and product ID, separated by an underscore character: `<MY VENDOR ID>_<MY PRODUCT ID>`.

  Insert your device specific information to this key. The key registers your device with Windows CE Plug-and-Play (USB driver) and enables identification of the device during boot. You can refer to the registry after calling **WDU_Init()** and then this key will exist. From that moment the device will be recognized by CE. If your device has a persistent registry, this addition will remain until you remove it.

  For more information, refer to MSDN Library, under *USB Driver Registry Settings* section.

6.4 Developing in Visual Basic and Delphi

The entire WinDriver API can be used when developing drivers in Visual Basic and Delphi.

6.4.1 Using DriverWizard

DriverWizard can be used to diagnose your hardware and verify that it is working properly before you start coding. You can then proceed to automatically generate source code with the wizard in a variety of languages, including Delphi and Visual Basic. For more information, refer to Chapter 5 and Section 6.4.3 below.

6.4.2 Samples

Samples for drivers written using the WinDriver API in Delphi or Visual Basic can be found in:

1. **WinDriver\delphi\samples**
2. **WinDriver\vb\samples**

Use these samples as a starting point for your own driver.

6.4.3 Creating your Driver

The method of development in Visual Basic is the same as the method in C using the automatic code generation feature of DriverWizard.

Your work process should be as follows:

- Use DriverWizard to easily diagnose your hardware.
- Verify that it is working properly.
- Generate your driver code.
- Integrate the driver into your application.
- You may find it useful to use the WinDriver samples to get to know the WinDriver API and as your skeletal driver code.

Chapter 7

Debugging Drivers

The following sections describe how to debug your hardware access application code.

7.1 User-Mode Debugging

- Since WinDriver is accessed from the user mode, we recommend that you first debug your code using your standard debugging software.
- The Debug Monitor utility [7.2] logs debug messages from WinDriver's kernel- and user-mode APIs. You can also use WinDriver APIs to send your own debug messages to the Debug Monitor log.
- Use DriverWizard to validate your device's USB configuration and test the communication with the device.

7.2 Debug Monitor

Debug Monitor is a powerful graphical- and console-mode tool for monitoring all activities handled by the WinDriver kernel (**windrvr6.sys/.dll/.o/.ko**).
You can use this tool to monitor how each command sent to the kernel is executed.
In addition, WinDriver enables you to print your own debug messages to the Debug Monitor, using the `WD_DebugAdd()` function [A.5.6] or the high-level `PrintDbgMessage()` function [A.6.14].

The Debug Monitor has two modes: graphical mode and console mode. The following sections explain how to operate Debug Monitor in both modes.

7.2.1 Using Debug Monitor in Graphical Mode

The graphical (GUI) version of the Debug Monitor utility is available for Windows 98/Me/2000/XP/Server 2003 and Linux. You may also use the Debug Monitor to debug your Windows CE driver code running on CE emulation on a Windows 2000/XP/Server 2003 platform. For Windows CE targets, use Debug Monitor in console mode [7.2.2].

1. Run the Debug Monitor using either of the following alternative methods:
   - Run **WinDriver/util/wddebug_gui**.
   - Run the Debug Monitor from the DriverWizard's **Tools** menu.
   - In Windows, use **Start | Programs | WinDriver | Debug Monitor** to start the Debug Monitor.

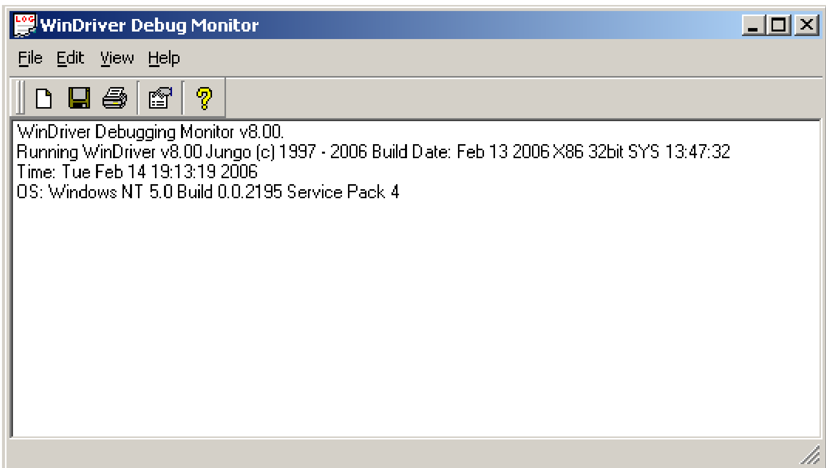


Figure 7.1: Start Debug Monitor

2. Set the Debug Monitor's status, trace level and debug sections information from the **Debug Options** dialogue, which is activated either from the Debug Monitor's **View | Debug Options** menu or the **Debug Options** toolbar button.

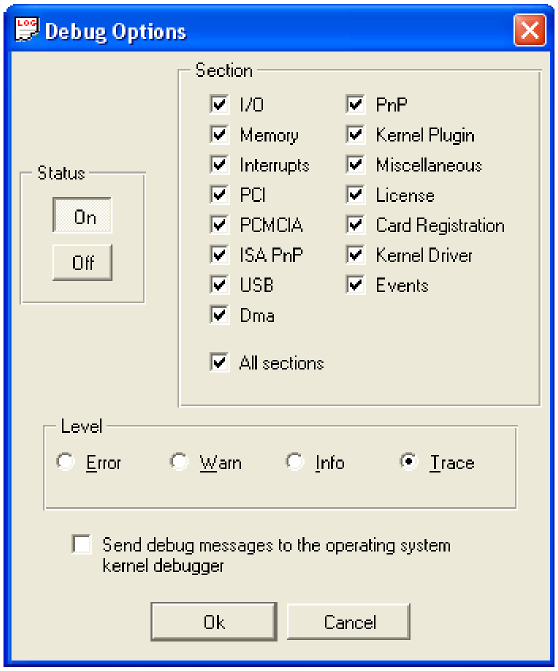


Figure 7.2: Debug Options

- **Status** – Set trace on or off.
- **Section** – Choose what part of the WinDriver API you would like to monitor. USB developers should select the **USB** section.

> **TIP**
> Choose carefully those sections that you would like to monitor. Checking more options than necessary could result in an overflow of information, making it harder for you to locate your problem.

- **Level** – Choose the level of messages you want to see for the resources defined.

  Error is the lowest trace level, resulting in minimum output to the screen.

Trace is the highest trace level, displaying every operation the WinDriver kernel performs.

- Select the **Send WinDriver Debug Messages To Kernel Debugger** option if you want debugging messages to be sent to an external kernel debugger as well.

  This option enables you to send to an external kernel debugger all the debug information that is received from WinDriver's kernel module.

  Now run your application, reproduce the problem, and view the debug information in the external kernel debugger's log.

  Windows users can use Microsoft's WinDbg tool, for example, which is freely supplied with Microsoft's Driver Development Kit (DDK) and from Microsoft's web site (Microsoft Debugging Tools page).

3. Once you have defined what you want to trace and on what level, click **OK** to close the **Debug Options** window.
4. Activate your program (step-by-step or in one run).
5. Watch the Debug Monitor log for errors or any unexpected messages.

7.2.2 Using Debug Monitor in Console Mode

The Debug Monitor utility comes in a console-mode version – **wddebug** – which is available for all supported operating systems.

7.2.2.1 Running wddebug on Windows, Windows CE and Linux

To use the **wddebug** console-mode version of the Debug Monitor utility on Windows 98/Me/2000/XP/Server 2003, Windows CE and Linux, run the following command from the **WinDriver/util/** directory:

```
wddebug [stat] [level] [sections]
```

NOTE
On Windows CE, start a Windows CE command window (**CMD.EXE**) on the Windows CE target and then run the program **WDDEBUG.EXE** inside this shell.

- **stat:** The Debug Monitor status –
  – **on**: Turn the Debug Monitor on.
  – **off**: Turn the Debug Monitor off.

  - **dbg_on**: Redirect the debug messages from the Debug Monitor to a kernel debugger and turn the Debug Monitor on.
  - **dbg_off**: Stop redirecting debug messages from the Debug Monitor to a kernel debugger.
  - **dump**: Continuously display debug information, until the user presses `ENTER`.
- **level:** The debug trace level – `ERROR`, `WARN`, `INFO` or `TRACE`, where `ERROR` is the lowest trace level and `TRACE` is the highest level (displays all messages).
- **section:** The debug sections, which determine what part of the WinDriver API you would like to monitor. For a full list of all supported debug sections, run **wddebug** with no parameters to view the program's usage instructions.

You can also run the command **`wddebug status`** to display the Debug Monitor version, the version of the running WinDriver driver module, and the current Debug Monitor status information (including the debug level and sections).

Normally, you should turn the Debug Monitor on with the desired debug level and section(s), then run **`wddebug dump`** to see the debug messages. When you are done, press `ENTER` to stop the debug dump and run **`wddebug off`** to turn off the Debug Monitor.

EXAMPLE

The following is an example of a typical **wddebug** usage sequence:

- Turn the Debug Monitor on with the highest trace level for all sections:
  `wddebug on TRACE ALL`
- Dump the debug messages continuously, until you hit ENTER:
  `wddebug dump`
- Turn the Debug Monitor off:
  `wddebug off`

Chapter 8

Enhanced Support for Specific Chipsets

8.1 Overview

In addition to the standard WinDriver API and the DriverWizard code generation capabilities described in this manual, which support development of drivers for any USB device, WinDriver offers enhanced support for specific USB chipsets. The enhanced support includes custom API and sample diagnostics code, which are designed specifically for these chipsets.

WinDriver's enhanced support is currently available for the following chipsets:
The Cypress EZ-USB family, Microchip PIC18F4550, Philips PDIUSBD12, Texas Instruments TUSB3410, TUSB3210, TUSB2136, TUSB5052, Silicon Laboratories C8051F320.

> **NOTE**
> The WinDriver USB Device toolkit's enhanced support for development of USB device firmware for the Cypress EZ-USB FX2LP CY7C68013A, Microchip PIC18F4550, Philips PDIUSBD12 and Silicon Laboratories C8051F320 and C8051F340 chipsets, is discussed separately in Chapter 12.

8.2 Developing a Driver Using the Enhanced Chipset Support

When developing a driver for a device based on one of the enhanced-support chipsets [8.1], you can use WinDriver's chipset-set specific support by following these steps:

1. Locate the sample diagnostics program for your device under the **WinDriver/chip_vendor/chip_name/** directory.

   Most of the sample diagnostics program names are derived from the sample's main purpose (e.g. **download_sample** for a firmware download sample) and their source code can be found directly under the specific **chip_name/** directory.

2. Run the custom diagnostics program to diagnose your device and familiarize yourself with the options provided by the sample program.

3. Use the source code of the diagnostics program as your skeletal device driver and modify the code, as needed, to suit your specific development needs. When modifying the code, you can utilize the custom WinDriver API for your specific chip. The custom API is typically found under the **WinDriver/chip_vendor/lib/** directory.

Chapter 9

USB Control Transfers

9.1 USB Control Transfers Overview

9.1.1 USB Data Exchange

The USB standard supports two kinds of data exchange between the host and the device:

Functional data exchange is used to move data to and from the device. There are three types of data transfers: Bulk, Interrupt, and Isochronous transfers.

Control exchange is used to configure a device when it is first attached and can also be used for other device-specific purposes, including control of other pipes on the device. Control exchange takes place via a control pipe, mainly the default Pipe 0, which always exists.

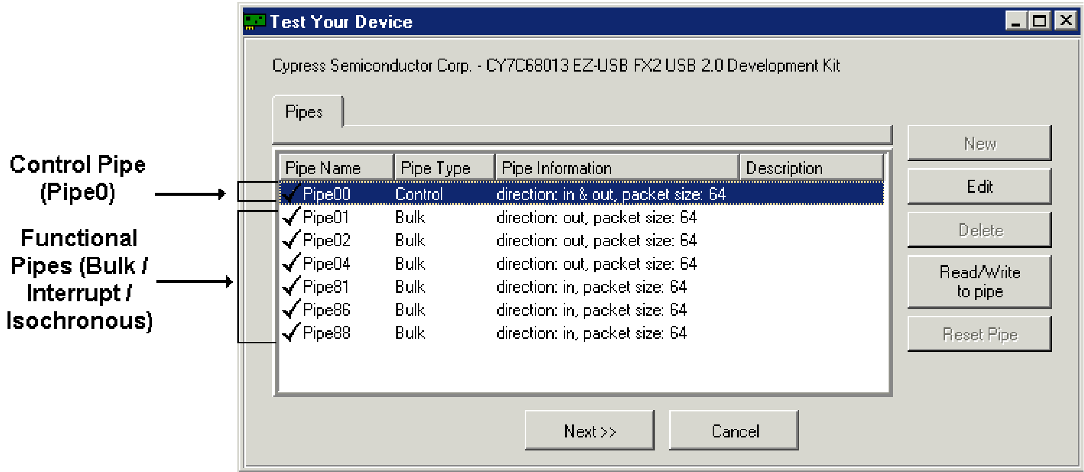


Figure 9.1: USB Data Exchange

9.1.2 More About the Control Transfer

The control transaction always begins with a setup stage. The setup stage is followed by zero or more control data transactions (data stage) that carry the specific information for the requested operation, and finally a status transaction completes the control transfer by returning the status to the host.

During the setup stage, an 8-byte setup packet is used to transmit information to the control endpoint of the device. The setup packet's format is defined by the USB specification.

A control transfer can be a read transaction or a write transaction. In a read transaction the setup packet indicates the characteristics and amount of data to be read from the device. In a write transaction the setup packet contains the command sent (written) to the device and the number of control data bytes that will be sent to the device in the data stage.

Refer to Figure 9.2 (taken from the USB specification) for a sequence of read and write transactions.
'(in)' indicates data flow from the device to the host.
'(out)' indicates data flow from the host to the device.

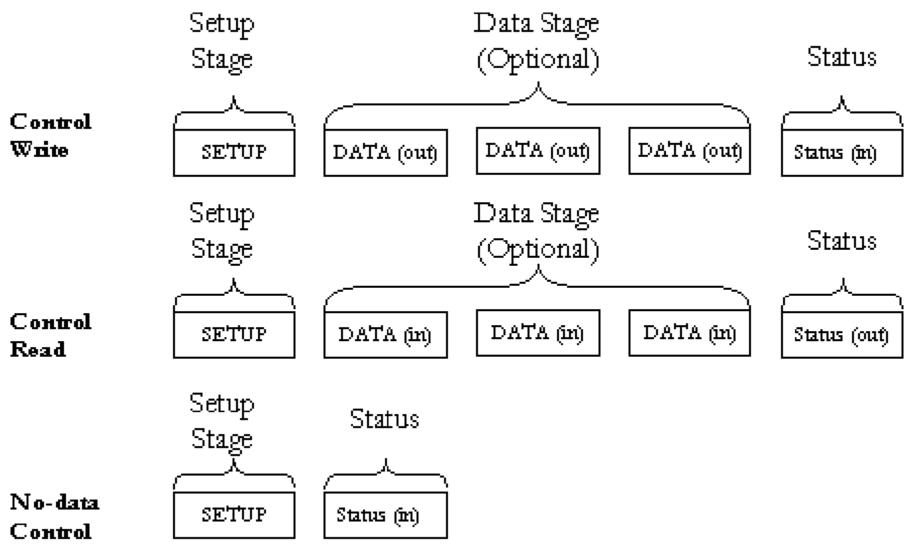


Figure 9.2: USB Read and Write

9.1.3 The Setup Packet

The setup packets (combined with the control data stage and the status stage) are used to configure and send commands to the device. Chapter 9 of the USB specification defines standard device requests. USB requests such as these are sent from the host to the device, using setup packets. The USB device is required to respond properly to these requests. In addition, each vendor may define device-specific setup packets to perform device-specific operations. The standard setup packets (standard USB device requests) are detailed below. The vendor's device-specific setup packets are detailed in the vendor's data book for each USB device.

9.1.4 USB Setup Packet Format

The table below shows the format of the USB setup packet. For more information, please refer to the USB specification at `http://www.usb.org`.

Byte	Field	Description
0	bmRequest Type	Bit 7: Request direction (0=Host to device – Out, 1=Device to host - In). Bits 5-6: Request type (0=standard, 1=class, 2=vendor, 3=reserved). Bits 0-4: Recipient (0=device, 1=interface, 2=endpoint,3=other).
1	bRequest	The actual request (see the Standard Device Request Codes table [9.1.5]).
2	wValueL	A word-size value that varies according to the request. For example, in the `CLEAR_FEATURE` request the value is used to select the feature, in the `GET_DESCRIPTOR` request the value indicates the descriptor type and in the `SET_ADDRESS` request the value contains the device address.
3	wValueH	The upper byte of the `Value` word.
4	wIndexL	A word-size value that varies according to the request. The index is generally used to specify an endpoint or an interface.
5	wIndexH	The upper byte of the `Index` word.
6	wLengthL	A word-size value that indicates the number of bytes to be transferred if there is a data stage.
7	wLengthH	The upper byte of the `Length` word.

9.1.5 Standard Device Request Codes

The table below shows the standard device request codes.

bRequest	**Value**
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

9.1.6 Setup Packet Example

This example of a standard USB device request illustrates the setup packet format and its fields. The setup packet is in Hex format.

The following setup packet is for a control read transaction that retrieves the device descriptor from the USB device. The device descriptor includes information such as USB standard revision, vendor ID and product ID.

GET_DESCRIPTOR (Device) Setup Packet

80	06	00	01	00	00	12	00

Setup packet meaning:

Byte	**Field**	**Value**	**Description**
0	BmRequest Type	80	8h=1000b bit 7=1 -> direction of data is from device to host. 0h=0000b bits 0..1=00 -> the recipient is the device.
1	bRequest	06	The Request is GET_DESCRIPTOR.
2	wValueL	00	
3	wValueH	01	The descriptor type is device (values defined in USB spec).
4	wIndexL	00	The index is not relevant in this setup packet since there is only one device descriptor.
5	wIndexH	00	
6	wLengthL	12	Length of the data to be retrieved: 18(12h) bytes (this is the length of the device descriptor).
7	wLengthH	00	

In response, the device sends the device descriptor data. A device descriptor of Cypress EZ-USB Integrated Circuit is provided as an example:

Byte No.	0	1	2	3	4	5	6	7	8	9	10
Content	12	01	00	01	ff	ff	ff	40	47	05	80

Byte No.	11	12	13	14	15	16	17
Content	00	01	00	00	00	00	01

As defined in the USB specification, byte 0 indicates the length of the descriptor, bytes 2-3 contain the USB specification release number, byte 7 is the maximum packet size for endpoint 00, bytes 8-9 are the Vendor ID, bytes 10-11 are the Product ID, etc.

9.2 Performing Control Transfers with WinDriver

WinDriver allows you to easily send and receive control transfers on Pipe00, while using DriverWizard to test your device. You can either use the API generated by DriverWizard [5] for your hardware, or directly call the WinDriver `WDU_Transfer()` [A.3.7] function from within your application.

9.2.1 Control Transfers with DriverWizard

1. Choose **Pipe00** and click **Read/Write To Pipe**.
2. You can either enter a custom setup packet, or use a standard USB request.
    - For a custom request: enter the required setup packet fields. For a write transaction that includes a data stage, enter the data in the **Write to pipe data (Hex)** field. Click **Read From Pipe** or **Write To Pipe** according to the required transaction (see Figure 9.3).

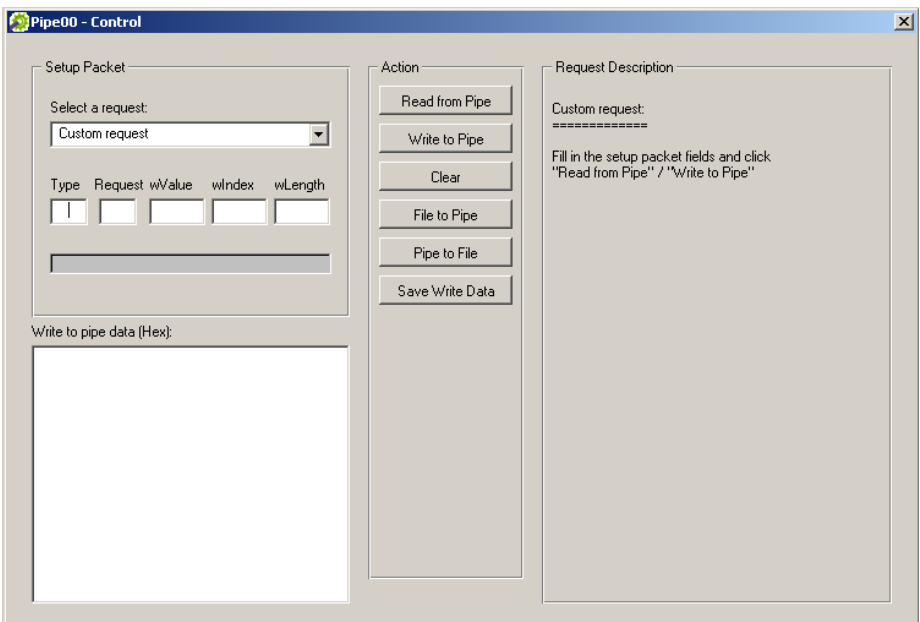


Figure 9.3: Custom Request

- For a standard USB request: select a USB request from the requests list, which includes requests such as **GET_DESCRIPTOR CONFIGURATION, GET_DESCRIPTOR DEVICE, GET_STATUS DEVICE**, etc. (see Figure 9.4). The description of the selected request will be displayed in the **Request Description** box on the right hand of the dialogue window.

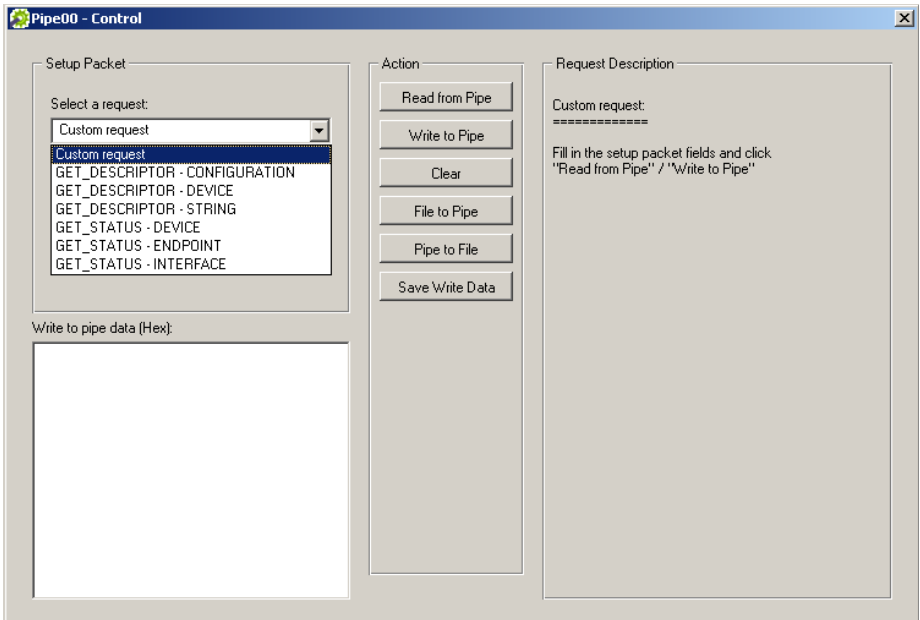


Figure 9.4: Request List

3. The results of the transfer, such as the data that was read or a relevant error, are displayed in Driver Wizard's **Log** window.
   Figure 9.5 below shows the contents of the **Log** window after a successful **GET_DESCRIPTOR DEVICE** request.

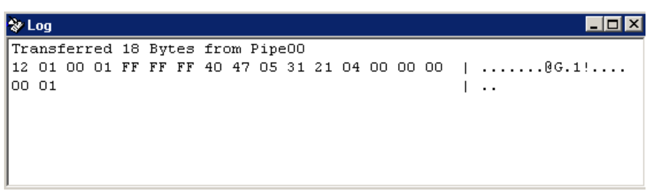


Figure 9.5: USB Request Log

9.2.2 Control Transfers with WinDriver API

To perform a read or write transaction on the control pipe, you can either use the API generated by DriverWizard for your hardware, or directly call the WinDriver `WDU_Transfer()` [A.3.7] function from within your application.

Fill the setup packet in the `BYTE SetupPacket[8]` array and call these functions to send setup packets on Pipe00 and to retrieve control and status data from the device.

- The following sample demonstrates how to fill the `SetupPacket[8]` variable with a `GET_DESCRIPTOR` setup packet:

```
setupPacket[0] = 0x80;   /* BmRequstType */
setupPacket[1] = 0x6;    /* bRequest  [0x6 == GET_DESCRIPTOR] */
setupPacket[2] = 0;      /* wValue */
setupPacket[3] = 0x1;    /* wValue [Descriptor Type: 0x1 == DEVICE] */
setupPacket[4] = 0;      /* wIndex */
setupPacket[5] = 0;      /* wIndex */
setupPacket[6] = 0x12;   /* wLength  [Size for the returned buffer] */
setupPacket[7] = 0;      /* wLength */
```

- The following sample demonstrates how to send a setup packet to the control pipe (a GET instruction; the device will return the information requested in the pBuffer variable):

```
WDU_TransferDefaultPipe(hDev, TRUE, 0, pBuffer, dwSize,
   bytes_transferred, &setupPacket[0], 10000);
```

- The following sample demonstrates how to send a setup packet to the control pipe (a SET instruction):

```
WDU_TransferDefaultPipe(hDev, FALSE, 0, NULL, 0,
    bytes_transferred, &setupPacket[0], 10000);
```

For further information regarding `WDU_TransferDefaultPipe()`, refer to section A.3.9. For further information regarding `WDU_Transfer()`, refer to section A.3.7.

Chapter 10

Dynamically Loading Your Driver

10.1 Why Do You Need a Dynamically Loadable Driver?

When adding a new driver, you may be required to reboot the system in order for it to load your new driver into the system. WinDriver is a dynamically loadable driver, which enables your customers to start your application immediately after installing it, without the need for reboot.

> **NOTE**
> In order to successfully UNLOAD your driver, make sure there are no open handles to the driver from WinDriver applications or from connected Plug and Play devices that were registered with WinDriver using an INF file.

10.2 Windows 98/Me/2000/XP/Server 2003

10.2.1 Windows Driver Types

Windows drivers can be implemented as either of the following types:

- WDM (Windows Driver Model) drivers: Files with the extension **.sys** on Windows 98/Me/2000/XP/Server 2003 (e.g. **windrvr6.sys**).
  WDM drivers are installed via the installation of an INF file (see below).

- Non-WDM / Legacy drivers: These include drivers for non-Plug and Play Windows operating systems (Windows NT 4.0) and files with the extension **.vxd** on Windows 98/Me.

The WinDriver Windows kernel module – **windrvr6.sys** – is a fully WDM driver, which can be installed using the **wdreg** utility, as explained in the following sections.

10.2.2 The WDREG Utility

WinDriver provides a utility for dynamically loading and unloading your driver, which replaces the slower manual process using Windows' Device Manager (which can still be used for the device INF). For **Windows 2000/XP/Server 2003**, this utility is provided in two forms: **wdreg** and **wdreg_gui**. Both versions can be found under the **WinDriver\util** directory, can be run from the command line, and provide the same functionality. The difference is that **wdreg_gui** displays installation messages graphically, while **wdreg** displays them in console mode.
For **Windows 98/Me**, the **wdreg16** utility is provided.
This section describes the usage of **wdreg**/ **wdreg_gui**/**wdreg16** on Windows operating systems.

NOTES

- The explanations and examples below refer to **wdreg**, but for **Windows 2000/XP/Server 2003** you can replace any references to **wdreg** with **wdreg_gui**.
  For **Windows 98/Me**, replace the references to **wdreg** with **wdreg16**.
- On **Windows 98/Me** you can only use **wdreg16** to install the **windrvr6.sys** WDM driver (by installing **windrvr6.inf**), but you **cannot** use **wdreg16** to install any other INF files.

This section explains how to use the **wdreg** utility to install the WDM **windrvr6.sys** driver on Windows 98/Me/2000/XP/Server 2003, or to install INF files that register USB devices to work with this driver on Windows 2000/XP/Server 2003.

Usage: The **wdreg** utility can be used in two ways as demonstrated below:

1. `wdreg -inf <filename> [-silent] [-log <logfile>] [install | uninstall | enable | disable]`
2. `wdreg -rescan <enumerator> [-silent] [-log <logfile>]`

- OPTIONS
  wdreg supports several basic OPTIONS from which you can choose one, some, or none:

  -inf – The path of the INF file to be dynamically installed.

  -rescan <enumerator> – Rescan enumerator (ROOT, USB, etc.) for hardware changes. Only one enumerator can be specified.

  -silent – Suppresses the display of messages of any kind (optional).

  -log <logfile> – Logs all messages to the specified file (optional).

- ACTIONS
  wdreg supports several basic ACTIONS:

  install – Installs the INF file, copies the relevant files to their target locations, dynamically loads the driver specified in the INF file name by replacing the older version (if needed).

  uninstall – Removes your driver from the registry so that it will not load on next boot (see note below).

  enable – Enables your driver.

  disable – Disables your driver, i.e. dynamically unloads it, but the driver will reload after system boot (see note below).

NOTE
In order to successfully disable/uninstall WinDriver, you must first close any open handles to the **windrvr6.sys** service. This includes closing any open WinDriver applications and uninstalling (from the Device Manager or using **wdreg**) any USB devices that are registered to work with the **windrvr6.sys** service (or otherwise removing such devices). **wdreg** will display a relevant warning message if you attempt to stop **windrvr6.sys** when there are still open handles to the service, and will enable you to select whether to close all open handles and Retry, or Cancel and reboot the PC to complete the command's operation.

10.2.3 Dynamically Loading/Unloading windrvr6.sys INF Files

When using WinDriver, you develop a user-mode application that controls and accesses your hardware by using the generic **windrvr6.sys** driver (WinDriver's kernel module). Therefore, you might want to dynamically load and unload the driver **windrvr6.sys** – which you can do using **wdreg**.
In addition, in WDM-compatible operating systems, you also need to dynamically load INF files for your Plug and Play devices. **wdreg** enables you to do so automatically on Windows 2000, XP and Server 2003.

This section includes **wdreg** usage examples, which are based on the detailed description of **wdreg** contained in the previous section.

- To start **windrvr6.sys** on Windows 98/Me/2000/XP/Server 2003:
  `wdreg -inf [path to windrvr6.inf] install`
  This command loads **windrvr6.inf** and starts the **windrvr6.sys** service.
- To load an INF file named **device.inf**, located under the **c:\tmp** directory, on Windows 2000/XP/Server 2003:
  `wdreg -inf c:\tmp\device.inf install`

To unload the driver/INF file, use the same commands, but simply replace *install* in the examples above with *uninstall*.

10.3 Linux

- To dynamically load WinDriver on Linux, execute:
  `/sbin/modprobe windrvr6`
- To dynamically unload WinDriver, execute:
  `/sbin/rmmod windrvr6`
- You can also use the **wdreg** script from the **WinDriver/util/** directory to install (load) **windrvr6.o/.ko**.

  `Usage:  wdreg <module name>`

  To install the **windrvr6** module run:
  `wdreg windrvr6`

> **TIP**
> To automatically load **windrvr6.o/.ko** on each boot, run the **wdreg** script from the target Linux **/etc/rc.d/rc.local** file:
> **`wdreg windrvr6`**

10.4 Windows Mobile 5.0

The **WinDriver****redist****Windows_Mobile_5_ARMV4I**\\ **wdreg.exe** utility can be used for loading the WinDriver kernel module (**windrvr6.dll**) on a Windows Mobile 5.0 platform.

> **TIP**
> On Windows Mobile 5.0 the operating system's security scheme prevents the loading of unsigned drivers at boot time, therefore the WinDriver kernel module has to be reloaded after boot. To load WinDriver on the target Windows Mobile 5.0 platform every time the OS is started, copy the **wdreg.exe** utility to the **Windows****StartUp**\\ directory on the target.

The source code of the Windows Mobile 5.0 **wdreg.exe** utility is available under the **WinDriver****samples****wince_install****wdreg**\\ directory on the development PC.

Chapter 11

Distributing Your Driver

Read this chapter in the final stages of driver development. It will guide you in preparing your driver for distribution.

NOTE
For **Windows 2000/XP/Server 2003**, all references to **wdreg** in this chapter can be replaced with **wdreg_gui**, which offers the same functionality but displays GUI messages instead of console-mode messages.
For **Windows 98/Me**, all references to **wdreg** should be replaced with **wdreg16**.
For more information regarding the **wdreg** utility, see Chapter 10.

11.1 Getting a Valid License for WinDriver

To purchase a WinDriver license, complete the **WinDriver/docs/order.txt** order form and fax or email it to Jungo. Complete details are included on the order form. Alternatively, you can order WinDriver on-line. For more details, visit `http://www.jungo.com`.

In order to install the registered version of WinDriver and to activate driver code that you have developed during the evaluation period on the development machine, please follow the installation instructions found in section 4.2 above.

11.2 Windows 98/Me/2000/XP/Server 2003

Distributing the driver you created is a multi-step process. First, create a distribution package that includes all the files required for the installation of the driver on the target computer. Second, install the driver on the target machine. This involves installing **windrvr6.sys** and **windrvr6.inf**, and installing the specific INF file for your device. Finally, you need to install and execute the hardware control application that you developed with WinDriver. These steps can be performed using **wdreg** utility.

11.2.1 Preparing the Distribution Package

Your distribution package should include the following files:

- Your hardware control application/DLL.
- **windrvr6.sys**.
  Get this file from the **WinDriver\redist** directory in the WinDriver package.
- **windrvr6.inf**.
  Get this file from the **WinDriver\redist** directory in the WinDriver package.
- **wd_api802.dll** (for distribution of 32-bit binaries to 32-bit target platforms or for distribution of 64-bit binaries to 64-bit platforms) or **wdapi802_32.dll** (for distribution of 32-bit binaries to 64-bit platforms).
  Get this file from the **WinDriver\redist** directory in the WinDriver package.
- An INF file for your device.
  You can generate this file with the DriverWizard, as explained in section 5.2.

11.2.2 Installing Your Driver on the Target Computer

> **NOTE**
> The user must have administrative privileges on the target computer in order to install your driver.

Follow the instructions below in the order specified to properly install your driver on the target computer:

- **Preliminary Steps:**
  - To avoid reboot, before attempting to install the driver make sure that there are no open handles to the **windrvr6.sys** service. This includes verifying that there are no open applications that use this service and that there are no connected Plug-and-Play devices that are registered to work with **windrvr6.sys** – i.e., no INF files that point to this driver are currently installed for any of the Plug-and-Play devices connected to the PC, or the INF file is installed but the device is disabled. This may be relevant, for example, when upgrading a driver developed with an earlier version of WinDriver (version 6.0 and later only, since previous versions used a different module name).
    You should therefore either disable or uninstall all Plug-and-Play devices that are registered to work with WinDriver from the Device Manager (**Properties | Uninstall**, **Properties | Disable** or **Remove** – on Win98/Me), or otherwise disconnect the device(s) from the PC. If you do not do this, attempts to install the new driver using **wdreg** will produce a message that instructs the user to either uninstall all devices currently registered to work with WinDriver, or reboot the PC in order to successfully execute the installation command.
  - On **Windows 2000**, remove any INF file(s) previously installed for your device (such as files created with an earlier version of WinDriver) from the **%windir%\inf** directory before installing the new INF file that you created for the device. This will prevent Windows from automatically detecting and installing an obsolete file. You can search the INF directory for the device's vendor ID and device/product ID to locate the file(s) associated with the device.

- **Install WinDriver's kernel module:**

  1. Copy **windrvr6.sys** and **windrvr6.inf** to the same directory.
  2. Use the utility **wdreg/wdreg16** to install WinDriver's kernel module on the target computer.

     On Windows 2000/XP/Server 2003, type from the command line:
     `wdreg -inf <path to windrvr6.inf> install`

     On Windows 98/Me, type from the command line:
     `wdreg16 -inf <path to windrvr6.inf> install`

     For example, if **windrvr6.inf** and **windrvr6.sys** are in the **d:\MyDevice** directory on the target computer, the command should be:
     `wdreg -inf d:\MyDevice\windrvr6.inf install`

     You can find the executable of **wdreg** in the WinDriver package under the **WinDriver\util** directory. For a general description of this utility and its usage, please refer to Chapter 10.

     > **NOTE**
     > **wdreg** is an interactive utility. If it fails, it will display a message instructing the user how to overcome the problem. In some cases the user may be asked to reboot the computer.

     > **CAUTION!**
     > When distributing your driver, take care not to overwrite a newer version of **windrvr6.sys** with an older version of the file in Windows drivers directory (**%windir%\system32\drivers**). You should configure your installation program (if you are using one) or your INF file so that the installer automatically compares the time stamp on these two files and does not overwrite a newer version with an older one.

- **Install the INF file for your device** (registering your Plug-and-Play device with **windrvr6.sys**):

  – **Windows 2000/XP/Server 2003**: Use the utility **wdreg** to automatically load the INF file.

    To automatically install your INF file on **Windows 2000/XP/Server 2003** and update Windows Device Manager, run **wdreg** with the **`install`** command:
    `wdreg -inf <path to your INF file> install`

> **NOTE**
> On **Windows 2000**, if another INF file was previously installed for the device, which registered the device to work with the Plug-and-Play driver used in earlier versions of WinDriver remove any INF file(s) for the device from the **%windir%\inf** directory before installing the new INF file that you created. This will prevent Windows from automatically detecting and installing an obsolete file. You can search the INF directory for the device's vendor ID and device/product ID to locate the file(s) associated with the device.

  – **Windows 98/Me**: Install the INF file manually using Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as outlined in detail in section 11.3 below.

- **Install wdapi802.dll:**
  If your hardware control application/DLL uses **wdapi802.dll** (as is the case for the sample and generated DriverWizard WinDriver projects), copy this DLL to the target's **%windir%\system32** directory.
  If you are distributing a 32-bit application/DLL to a target 64-bit platform, rename **wdapi802_32.dll** to **wdapi802.dll** and copy this file to the target's **%windir%\sysWOW64** directory.

> **NOTE**
> If you attempt to write a 32-bit installation program that installs a 64-bit program, and therefore copies the 64-bit **wdapi802.dll** DLL to the **%windir%\system32** directory, you may find that the file is actually copied to the 32-bit **%windir%\sysWOW64** directory. The reason for this is that Windows x64 platforms translate references to 64-bit directories from 32-bit commands into references to 32-bit directories. You can avoid the problem by using 64-bit commands to perform the necessary installation steps from your 32-bit installation program. The **system64.exe** program, provided in the **WinDriver\redist** directory of the Windows x64 WinDriver distributions, enables you to do this.

- **Install your hardware control application/DLL**: Copy your hardware control application/DLL to the target and run it!

11.3 Creating an INF File

Device information (INF) files are text files that provide information used by the Plug-and-Play mechanism in Windows 98/Me/2000/XP/Server 2003 to install software that supports a given hardware device. INF files are required for hardware that identifies itself, such as USB and PCI. An INF file includes all necessary information about a device and the files to be installed. When hardware manufacturers introduce new products, they must create INF files to explicitly define the resources and files required for each class of device.

In some cases, the INF file for your specific device is supplied by the operating system. In other cases, you will need to create an INF file for your device. WinDriver's DriverWizard can generate a specific INF file for your device. The INF file is used to notify the operating system that WinDriver now handles the selected device.

For USB devices, you will not be able to access the device with WinDriver (either from the DriverWizard or from the code) without first registering the device to work with **windrvr6.sys**. This is done by installing an INF file for the device. The DriverWizard will offer to automatically generate the INF file for your device.

You can use the DriverWizard to generate the INF file on the development machine – as explained in section 5.2 of the manual – and then install the INF file on any machine to which you distribute the driver, as explained in the following sections.

11.3.1 Why Should I Create an INF File?

- To stop the Windows **Found New Hardware Wizard** from popping up after each boot.
- To enable the DriverWizard to access USB devices.
- To ensure that the operating system can assign physical addresses to a USB device.
- To load the new driver created for the device.
  An INF file must be created whenever developing a new driver for Plug and Play hardware that will be installed on a Plug-and-Play system.
- To replace the existing driver with a new one.

11.3.2 How Do I Install an INF File When No Driver Exists?

> **NOTE**
> You must have administrative privileges in order to install an INF file on Windows 98/Me/2000/XP/Server 2003.

- **Windows 2000/XP/Server 2003**:

  On Windows 2000/XP/Server 2003 you can use the **wdreg** utility with the **`install`** command to automatically install the INF file:

  ```
  wdreg -inf <path to the INF file> install
  ```

  (for more information, refer to section 10.2.2 of the manual).

  On the development PC, you can have the INF file automatically installed when selecting to generate the INF file with the DriverWizard, by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation window (see section 5.2).

  It is also possible to install the INF file manually on Windows 2000/XP/Server 2003, using either of the following methods:

  – Windows **Found New Hardware Wizard**: This wizard is activated when the device is plugged in or, if the device was already connected, when scanning for hardware changes from the Device Manager.

  – Windows **Add/Remove Hardware Wizard**: Right-click the mouse on **My Computer**, select **Properties**, choose the **Hardware** tab and click on **Hardware Wizard...**.

  – Windows **Upgrade Device Driver Wizard**: Select the device from the **Device Manager** devices list, select **Properties**, choose the **Driver** tab and click the **Update Driver...** button. On Windows XP and Windows Server 2003 you can choose to upgrade the driver directly from the Properties list.

  In all the manual installation methods above you will need to point Windows to the location of the relevant INF file during the installation.
  We recommend using the **wdreg** utility to install the INF file automatically, instead of installing it manually.

- **Windows 98/Me**:

  On **Windows 98/Me** you need to install the INF file for your USB device manually, either via Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as explained below:

  – Windows **Add New Hardware Wizard**:

> **NOTE**
> This method can be used if no other driver is currently installed for the device or if the user first uninstalls (removes) the current driver for the device. Otherwise, Windows **New Hardware Found Wizard**, which activates the **Add New Hardware Wizard**, will not appear for this device.

(1) To activate the Windows **Add New Hardware Wizard**, attach the hardware device to the computer or, if the device is already connected, scan for hardware changes (**Refresh**).

(2) When Windows **Add New Hardware Wizard** appears, follow its installation instructions. When asked, point to the location of the INF file in your distribution package.

– Windows **Upgrade Device Driver Wizard**:

(1) Open Windows Device Manager: From the **System Properties** window (right-click on **My Computer** and select **Properties**) select the **Device Manager** tab.

(2) Select your device from the **Device Manager** devices list, choose the **Driver** tab and click the **Update Driver** button.
To locate your device in the Device Manager, select **View devices by connection** and navigate to **Standard PC | PCI bus | PCI to USB Universal Host Controller (or any other controller you are using – OHCI/EHCI) | USB Root Hub | <your device>**.

(3) Follow the instructions of the **Upgrade Device Driver Wizard** that opens. When asked, point to the location of the INF file in your distribution package.

11.3.3 How Do I Replace an Existing Driver Using the INF File?

> **NOTE**
> You must have administrative privileges in order to replace a driver on Windows 98/Me/2000/XP/Server 2003.

1. On **Windows 2000**, if you wish to upgrade the driver for USB devices that have been registered to work with earlier versions of WinDriver, we recommend that you first delete from Windows INF directory (**%windir%\inf**) any previous INF files for the device, to prevent Windows from installing an old INF file in place of the new file that you created. Look for files containing your device's vendor and device IDs and delete them.

2. Install your INF file:

   - On **Windows 2000/XP/Server 2003** you can automatically install the INF file:

     You can use the **wdreg** utility with the `install` command to automatically install the INF file on Windows 2000/XP/Server 2003:

     ```
     wdreg -inf <path to INF file> install
     ```

     (for more information, refer to section 10.2.2 of the manual).

     On the development PC, you can have the INF file automatically installed when selecting to generate the INF file with the DriverWizard, by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation window (see section 5.2).

     It is also possible to install the INF file manually on Windows 2000/XP/Server 2003, using either of the following methods:

     – Windows **Found New Hardware Wizard**: This wizard is activated when the device is plugged in or, if the device was already connected, when scanning for hardware changes from the Device Manager.

     – Windows **Add/Remove Hardware Wizard**: Right-click on **My Computer**, select **Properties**, choose the **Hardware** tab and click on **Hardware Wizard...**.

     – Windows **Upgrade Device Driver Wizard**: Select the device from the **Device Manager** devices list, select **Properties**, choose the **Driver** tab and click the **Update Driver...** button. On Windows XP and Windows Server 2003 you can choose to upgrade the driver directly from the Properties list.

     In the manual installation methods above you will need to point Windows to the location of the relevant INF file during the installation. If the installation wizard offers to install an INF file other than the one you have generated, select **Install one of the other drivers** and choose your specific INF file from the list.

     We recommend using the **wdreg** utility to install the INF file automatically, instead of installing it manually.

   - On **Windows 98/Me** you need to install the INF file manually via Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as explained below:

     – Windows **Add New Hardware Wizard**:

> **NOTE**
> This method can be used if no other driver is currently installed for the device or if the user first uninstalls (removes) the current driver for the device. Otherwise, the Windows **Found New Hardware Wizard**, which activates the **Add New Hardware Wizard**, will not appear for this device.

(1) To activate the Windows **Add New Hardware Wizard**, attach the hardware device to the computer or, if the device is already connected, scan for hardware changes (Refresh).

(2) When Windows **Add New Hardware Wizard** appears, follow its installation instructions. When asked, specify the location of the INF file in your distribution package.

– Windows **Upgrade Device Driver Wizard**:

(1) Open Windows Device Manager: From the **System Properties** window (right click on **My Computer** and select **Properties**) select the **Device Manager** tab.

(2) Select your device from the **Device Manager** devices list, open it, choose the **Driver** tab and click the **Update Driver** button. To locate your device in the Device Manager, select **View devices by connection** and navigate to **Standard PC | PCI bus | PCI to USB Universal Host Controller (or any other controller you are using – OHCI/EHCI) | USB Root Hub | <your device>**.

(3) Follow the instructions of the **Upgrade Device Driver Wizard** that opens. Locate the INF in your distribution package when asked.

11.4 Windows CE / Windows Mobile 5.0

11.4.1 Distribution to New Windows CE Platforms

To distribute the driver you developed with WinDriver to a new target Windows CE platform, follow these steps:

1. If you have not already done so, edit the project registry file to match your target hardware. If you select to use the WinDriver component, as outlined in step 2, the registry file to modify is **WinDriver\samples\wince_install \<TARGET_CPU>\WinDriver.reg** (e.g. **WinDriver\samples\wince_install\ARMV4I\ WinDriver.reg**). Otherwise, modify the **WinDriver\samples\wince_install\project_wd.reg** file.

2. You can simplify the driver integration into your Windows CE platform by following the procedure described in this step before the Sysgen platform compilation stage.

   NOTE:

   - This procedure provides a convenience method for integrating WinDriver into your Windows CE platform. If you select not to use this method, you will need to perform the manual integration steps described in step 4 below after the Sysgen stage.

   - The procedure described in this step also adds the WinDriver kernel module (**windrvr6.dll**) to your OS image. This is a necessary step if you want the WinDriver CE kernel file (**windrvr6.dll**) to be a permanent part of the Windows CE image (**NK.BIN**), which is the case if you select to transfer the file to your target platform using a floppy disk. However, if you prefer to have the file **windrvr6.dll** loaded on demand via the CESH/PPSH services, you need to perform the manual integration method described in step 4 instead of performing the procedure described in the present step.

   (a) Run Microsoft **Platform Builder** and open your platform.

   (b) From the **File** menu select **Manage Catalog Items....** and then click the **Import...** button and select the **WinDriver.cec** file from the relevant **WinDriver\samples\wince_install\<TARGET_CPU>** directory (e.g. **WinDriver\samples\wince_install\ARMV4I**).
   This will add a WinDriver component to the Platform Builder Catalog.

   (c) In the **Catalog** view, right-click the mouse on the **WinDriver Component** node in the **Third Party** tree and select **Add to OS design**.

3. Compile your Windows CE platform (Sysgen stage).
4. If you have chosen not to perform the procedure described in step 2 above, perform the following steps after the Sysgen stage in order to manually integrate the driver into your platform.

   NOTE: If you followed the procedure described in step 2, skip this step and go directly to step 5.

   (a) Run Microsoft **Platform Builder** and open your platform.

   (b) Select **Open Build Release Directory** from the **Build** menu.

   (c) Copy the WinDriver CE kernel file – **WinDriver\redist\<TARGET_CPU>\windrvr6.dll –** to the **%_FLATRELEASEDIR%** sub-directory on the target development platform (should be the current directory in the new command window).

   (d) Append the contents of the **project_wd.reg** file in the **WinDriver\samples\wince_install** directory to the **project.reg** file in the **%_FLATRELEASEDIR%** sub-directory.

   (e) Append the contents of the **project_wd.bib** file in the **WinDriver\samples\wince_install** directory to the **project.bib** file in the **%_FLATRELEASEDIR%** sub-directory.

   This step is only necessary if you want the WinDriver CE kernel file (**windrvr6.dll**) to be a permanent part of the Windows CE image (**NK.BIN**), which is the case if you select to transfer the file to your target platform using a floppy disk. If you prefer to have the file **windrvr6.dll** loaded on demand via the CESH/PPSH services, you do not need to carry out this step until you build a permanent kernel.
5. Select **Make Image** from the **Build** menu and name the new image **NK.BIN**.
6. Download your new kernel to the target platform and initialize it either by selecting **Download/Initialize** from the **Target** menu or by using a floppy disk.
7. Restart your target CE platform. The WinDriver CE kernel will automatically load.
8. Install your hardware control application/DLL on the target.
   If your hardware control application/DLL uses **wdapi802.dll** (as is the case for the sample and generated DriverWizard WinDriver projects), also copy this DLL from the **WinDriver\redist\WINCE\<TARGET_CPU>** directory on the Windows host development PC to the target's **Windows** directory.

11.4.2 Distribution to Windows CE / Windows Mobile 5.0 Computers

> **NOTE**
> Unless otherwise specified, ”Windows CE” references in this section include Windows CE 4.x – 5.x and Windows Mobile 5.0.

1. Copy WinDriver’s kernel module – **windrvr6.dll** – from the **WinDriver\redist\WINCE\<TARGET_CPU>** directory on the Windows host development PC to the **Windows** directory on your target Windows CE platform.

2. Add WinDriver to the list of device drivers Windows CE loads on boot:

   - Modify the registry according to the entries documented in the file **WinDriver\samples\wince_install\ project_wd.reg**. This can be done using the Windows CE Pocket Registry Editor on the hand-held CE computer or by using the Remote CE Registry Editor Tool supplied with MS eMbedded Visual C++ (Windows CE 4.x – 5.x) / MSDEV .NET 2005 (Windows Mobile 5.0). Note that in order to use the Remote CE Registry Editor tool you will need to have Windows CE Services installed on your Windows host platform.

   - On Windows Mobile 5.0 the operating system’s security scheme prevents the loading of unsigned drivers at boot time, therefore the WinDriver kernel module has to be reloaded after boot. To load WinDriver on the target Windows Mobile 5.0 platform every time the OS is started, copy the **WinDriver\redist\Windows_Mobile_5_ARMV4I\ wdreg.exe** utility to the **Windows\StartUp** directory on the target.

3. Restart your target CE computer. The WinDriver CE kernel will automatically load. You will have to do a warm reset rather than just suspend/resume (use the reset or power button on your target CE computer).

4. Install your hardware control application/DLL on the target.
   If your hardware control application/DLL uses **wdapi802.dll** (as is the case for the sample and generated DriverWizard WinDriver projects), also copy this DLL from the **WinDriver\redist\WINCE\<TARGET_CPU>** directory on the development PC to the target’s **Windows** directory.

11.5 Linux

The Linux kernel is continuously under development and kernel data structures are subject to frequent changes. To support such a dynamic development environment and still have kernel stability, the Linux kernel developers decided that kernel modules must be compiled with header files identical to those with which the kernel itself was compiled. They enforce this by including a version number in the kernel header files that is checked against the version number encoded into the kernel. This forces Linux driver developers to facilitate recompilation of their driver based on the target system's kernel version.

11.5.1 WinDriver Kernel Module

Since **windrvr6.o/.ko** is a kernel module, it must be recompiled for every kernel version on which it is loaded. To facilitate this, we supply the following components to insulate the WinDriver kernel module from the Linux kernel:

- **windrvr_gcc_v2.a**, **windrvr_gcc_v3.a** and **windrvr_gcc_v3_regparm.a**: compiled object code for the WinDriver kernel module. **windrvr_gcc_v2.a** is used for kernels compiled with GCC v2.x.x, and **windrvr_gcc_v3.a** is used for kernels compiled with GCC v3.x.x. **windrvr_gcc_v3_regparm.a** is used for kernels compiled with GCC v3.x.x with the **regparm** flag.
- **linux_wrappers.c/h**: wrapper library source code files that bind the WinDriver kernel module to the Linux kernel.
- **linux_common.h**, **windrvr.h**, **wd_ver.h** and **wdusb_interface.h**: header files required for building the WinDriver kernel module on the target.
- **wdusb_linux.c**: used by WinDriver to utilize the USB stack.
- **configure**: a configuration script, which creates a **makefile** that compiles and inserts the module **windrvr6.o/.ko** into the kernel.
- **makefile.in**, **wdreg** and **setup_inst_dir**: the **configure** script uses **makefile.in**, which creates a makefile. This makefile calls the **wdreg** utility shell script and **setup_inst_dir**, which we supply under the **WinDriver/util** directory. All three must be copied to the target.

> **TIP**
> You can use the **wdreg** script to load the WinDriver kernel module [10.3].
> To automatically load **windrvr6.o/.ko** on each boot, run the **wdreg** script from the target Linux **/etc/rc.d/rc.local** file:
> `wdreg windrvr6`

You need to distribute these components along with your driver source/object code.

11.5.2 User-Mode Hardware Control Application/Shared Objects

Copy the hardware control application/shared objects that you created with WinDriver to the target.

If your hardware control application/shared objects use **libwdapi802.so** (as is the case for the sample and generated DriverWizard WinDriver projects), copy this shared object from the **WinDriver/lib/** directory on the development PC to the target's library directory (**/usr/lib/** – for 32-bit PowerPC, 32-bit x86 targets or IA64 targets; **/usr/lib64/** – for 64-bit x86 targets).

Since your hardware control application/shared objects do not have to be matched against the kernel version number, you are free to distribute it as binary code (if you wish to protect your source code from unauthorized copying) or as source code. Note that under the license agreement with Jungo you may not distribute the source code of the **libwdapi802.so** shared object.

> **CAUTION!**
> If you select to distribute your source code, make sure you do not distribute your WinDriver license string, which is used in the code.

11.5.3 Installation Script

We suggest that you supply an installation shell script that copies your driver executables/DLL to the correct locations (perhaps **/usr/local/bin**) and then invokes **make** or **gmake** to build and install the WinDriver kernel module.

Chapter 12

WinDriver USB Device

This chapter describes the WinDriver USB Device tool-kit for development of USB device firmware for devices based on the Cypress EZ-USB FX2LP CY7C68013A, Microchip PIC18F4550, Philips PDIUSBD12 and Silicon Laboratories C8051F320 and C8051F340 hardware.

NOTE The WinDriver USB Device tool-kit is currently only supported on Windows – see section 12.2 for details regarding the supported operating systems.

12.1 WinDriver USB Device Overview

The WinDriver USB Device tool-kit simplifies and facilitates the development of firmware for USB devices based on the **Cypress EZ-USB FX2LP CY7C68013A, Microchip PIC18F4550, Philips PDIUSBD12 and Silicon Laboratories C8051F320 and C8051F340** hardware, which will henceforth be referred to in this chapter as **”the target hardware”**.

This tool-kit complements the WinDriver USB Driver Development Kit. Together these tool-kits provide a complete USB device development software solution – both for the device firmware and the host driver development stages.

USB device manufacturers need to support the Universal Serial Bus (USB) specification (see Chapter 3 for an overview of USB). The USB interface is implemented in two levels: The lower level of the USB protocol is implemented via a Serial Interface Engine (SIE). The higher layer of the protocol is implemented via the device firmware.

Firmware consists of software programs and data that define the device's configuration and are installed semi-permanently into memory using various types of programmable ROM chips, such as PROMS, EPROMs, EEPROMs, and flash chips.

WinDriver USB Device enables developers of devices based on the target hardware to easily create firmware that defines the desired USB interface for their target device, using a Graphical User Interface (GUI).

WinDriver USB Device includes **firmware libraries** for the target hardware [12.3.5]. These libraries contain functions for performing common USB firmware functionality, thus releasing device manufacturers of the time-consuming effort of writing this firmware code themselves.
For the **Microchip PIC18F4550** development board, the tool-kit also includes a special **Mass Storage** firmware library, which provides special mass storage APIs that comply with the **USB Mass Storage Class Specification** (`http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf`) and **USB Mass Storage Bulk-Only Transport Specification** (`http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf`).

WinDriver USB Device features the graphical **DriverWizard** utility from the WinDriver USB Driver Development Kit, but with different functionality, which enables you to **define your device's USB interface** [12.4.1] – i.e. the device IDs and device class, the number of interfaces, alternate settings and endpoints and their attributes, etc. – using friendly GUI dialogues, and then proceed to **generate firmware code** for the device, based on the information defined in the wizard's dialogues [12.4.2]. The generated DriverWizard firmware code includes convenient APIs, which utilize the WinDriver USB Device firmware library APIs to implement a fully functional device firmware.

For the **Microchip PIC18F4550** development board, DriverWizard features a special GUI dialogue for defining a **Mass Storage** USB device (see section 12.4.1 step # 4). When using this option, the generated firmware code includes convenient APIs, which utilize the WinDriver USB Device Microchip PIC18F4550 mass storage firmware library APIs to implement mass storage device firmware.

Appendices B – E provide a detailed description of the WinDriver USB Device firmware libraries and generated DriverWizard API.

In addition, the tool-kit includes **firmware samples** for the target hardware, which demonstrate how to utilize the library APIs for different purposes [12.3].

> **NOTE**
> The provided APIs and the wizard options for your target hardware are based on Chapter 9 of the USB 2.0 Specification and on the target hardware's specification, thus freeing you of the need to study these specifications yourself.
> The Microchip PIC18F4550 mass storage APIs and wizard options free you of the need to study the USB Mass Storage Class Specification.

After generating the firmware code, you can proceed to modify it, as needed, in order to implement your desired firmware, using the WinDriver USB Device API to simplify the development process [12.4.3]. When the firmware implementation is completed, you can simply build the firmware [12.4.3.2] and download it to the device [12.4.3.3].

The hardware diagnostics feature of the WinDriver USB Driver Development Kit's DriverWizard, as outlined in Chapter 5, is also available in the WinDriver USB Device DriverWizard. Therefore, once you develop the firmware and download it to the device, you can use DriverWizard to **debug the hardware** by viewing the device's configuration and testing the communication with the device from the wizard's graphical interface [12.4.4].

If you are also a registered user of the WinDriver USB Driver Development Kit, when the device firmware development and the hardware debugging is completed, you can use the WinDriver USB tool-kit to **develop a driver** for your device [12.4.5].

12.2 System and Hardware Requirements

To use the WinDriver USB Device tool-kit, the following host system and hardware requirements must be filled:

- Operating System: Windows 98/Me/2000/XP/Server 2003.

  To compile and build the **Cypress EZ-USB FX2LP CY7C68013A, Microchip PIC18F4550** and **Silicon Laboratories C8051F320 and C8051F340** firmware code you need Windows 2000/XP/Server 2003.

  To compile and build the **Philips PDIUSBD12** firmware code you need Windows 98/Me/2000/XP/Server 2003 or DOS.
  To download the PDIUSBD12 evaluation firmware to the device you need DOS.

- CPU architecture: Any x86 32-bit or 64-bit (x64: AMD64 or Intel EM64T) processor.

- The following development tools must be installed on your development PC in order to build the sample and generated firmware code:

  – For the **Cypress EZ-USB FX2LP CY7C68013A** hardware:
    The Cypress EZ-USB FX2LP Development Kit.

  – For the **Microchip PIC18F4550** hardware:
    The Microchip MCC18 Toolchain.

  – For the **Cypress EZ-USB FX2LP CY7C68013A** and **Silicon Laboratories C8051F320 and C8051F340** hardware:
    The Keil Cx51 development tools for 8x51, version 6.0 or above.

  – For the **Philips PDIUSBD12** hardware:
    A 32-bit DOS C compiler.

- The sample and generated firmware code support the following optional development environments:

  – For the **Cypress EZ-USB FX2LP CY7C68013A** and **Silicon Laboratories C8051F320 and C8051F340** hardware:
    The Keil μVision IDE, version 2.0 or above.

  – For the **Microchip PIC18F4550** hardware:
    The Microchip MPLAB IDE, version 7.20.

  – For the **Silicon Laboratories C8051F320 and C8051F340** hardware:
    The Silicon Laboratories IDE, version 1.9.

  – For the **Philips PDIUSBD12** hardware:
    The Borland C++, version 3.1, 32-bit DOS compiler (**”Turbo C”**).

12.3 WinDriver Device Firmware (WDF) Directory Overview

This section describes the directory structure and files of the **WinDriver\wdf** directory.

The **wdf** directory contains the following sub-directories:

- **cypress** directory: Contains files for devices based on the Cypress EZ-USB FX2LP CY7C68013A development board.
- **microchip** directory: Contains files for devices based on the Microchip PIC18F4550 development board.
- **philips** directory: Contains files for devices based on the Philips PDIUSBD12 hardware.
- **silabs** directory: Contains files for devices based on the Silicon Laboratories C8051F320 and C8051F340 development boards.

12.3.1 The cypress Directory

The **WinDriver\wdf\cypress** directory contains the following directories:

- **FX2LP** directory: Contains files for devices based on the FX2LP CY7C68013A development board (henceforth in this section – **”the FX2LP board”**).

The **FX2LP** directory contains the following sub-directories and files:

- **include** directory:
  - **wdf_cypress_lib.h**: Header file that contains firmware library types, general definitions and function prototypes for devices based on the FX2LP board. This file provides the interface of the board’s firmware library (**wdf_cypress_fx2lp_eval.lib** – for evaluation users; For registered users the library’s source code is created as part of the DriverWizard device firmware code generation – see explanation regarding the WinDriver USB device firmware libraries in section 12.3.5).
  - **wdf_cypress.h**: Header file that contains the required firmware libraries definitions and #include statements for utilizing the Cypress FX2LP API.
  - **periph.h**: Header file that contains function prototypes for supporting USB peripheral device functionality for devices based on the FX2LP board. The functions’ implementation is dependent on the specific configuration defined for the device. The **periph.c** source file

that contains the implementation for your device is created by the DriverWizard when generating device firmware code, based on the USB device configuration that you define in the wizard see the description of the generated DriverWizard files [12.4.3.1].

- **lib**\ directory:
  - **wdf_cypress_fx2lp_eval.lib**: Evaluation firmware library for the FX2LP board (see explanation below [12.3.5]).
- **samples**\ directory: Device firmware samples for the FX2LP board:
  - **loopback**\ directory: Loopback sample: The sample implements a loopback, which fills the OUT endpoint's FIFO buffer with the data that is read from the IN endpoint's FIFO buffer.
    - **periph.c**: Source file that contains sample implementation of the functions declared in the **periph.h** header file (discussed above).
    - **wdf_dscr.a51**: Assembly file that contains sample descriptor data tables definitions for the FX2LP board.
    - **build.bat**: A utility for building the sample firmware code.
      Note: The build utility uses the evaluation firmware library (**wdf_cypress_fx2lp_eval.lib**).
    - **loopback_eval.hex**: Sample loopback firmware for the FX2LP board, created by building the sample code with the **build.bat** utility. **Note:** The firmware uses the evaluation firmware library (**wdf_cypress_fx2lp_eval.lib**).

12.3.2 The microchip Directory

The **WinDriver****wdf****microchip** directory contains the following directories:

- **18F4550**\ directory: Contains files for devices based on the PIC18F4550 development board.

The **18F4550**\ directory contains the following sub-directories and files:

- **include**\ directory:
  - **class****msd**\ directory: Contains USB mass storage device class firmware API declarations and type definitions for the PIC18F4550 board:
    - **wdf_msd.h**: Header file that contains mass storage firmware library types, general definitions and function prototypes for the PIC18F4550 board.

* **wdf_disk.h**: Header file that contains types, general definitions and function prototypes for accessing the storage media on mass storage devices based on the PIC18F4550 board.
  Note: The implementation of the functions declared in this header is hardware-specific. The generated DriverWizard **wdf_xxx_hw.c** file [12.4.3.1] contains implementation stubs for these functions. The **WinDriver\wdf\microchip\18F4550\samples\msd\sdcard.c** file (described below) contains a sample implementation of the storage media access functions for an SD Card.

– **wdf_microchip_lib.h**: Header file that contains general firmware library types, definitions and function prototypes for devices based on the PIC18F4550 board. This file, together with the **wdf_usb9.h** header and the **wdf_msd.h** header (when developing mass storage firmware), provides the interface of the board's firmware library (**wdf_microchip_18f4550_eval.lib** / **wdf_microchip_msd_18f4550_eval.lib** (mass storage) – for evaluation users; For registered users the library's source code is created as part of the DriverWizard device firmware code generation – see explanation regarding the WinDriver USB device firmware libraries in section 12.3.5).

– **wdf_usb9.h**: Header file that contains firmware library USB descriptors type definitions and function declarations based on Chapter 9 of the USB 2.0 Specification.

– **wdf_microchip.h**: Header file that contains general firmware library definitions for the PIC18F4550 board. This header includes all other required header files for the PIC18F4550 board, therefore when developing firmware for this board you need only include this header from your source files.

– **types.h**: Header file that defines data types for the PIC18F4550 board.

– **periph.h**: Header file that contains function prototypes for supporting USB peripheral device functionality for devices based on the PIC18F4550 board. The functions' implementation is dependent on the specific configuration defined for the device. The **periph.c** source file that contains the implementation for your device is created by the DriverWizard when generating device firmware code, based on the USB device configuration that you define in the wizard see the description of the generated DriverWizard files [12.4.3.1].

- **lib**\ directory:
  - **wdf_microchip_18f4550_eval.lib**: Evaluation firmware library for the PIC18F4550 board, compatible with MCC18 version 3.0.
  - **wdf_microchip_18f4550_eval_v24.lib**: Evaluation firmware library for the PIC18F4550 board, compatible with MCC18 version 2.4.
  - **wdf_microchip_msd_18f4550_eval.lib**: Evaluation mass storage firmware library for the PIC18F4550 board, compatible with MCC18 version 3.0.
  - **wdf_microchip_msd_18f4550_eval_v24.lib**: Evaluation mass storage firmware library for the PIC18F4550 board, compatible with MCC18 version 2.4.

  For details regarding the firmware libraries, refer to section 12.3.5.
- **samples**\ directory: Device firmware samples for the PIC18F4550 board:
  - **loopback**\ directory: Loopback sample: The sample implements a loopback, which fills the OUT endpoint's FIFO buffer with the data that is read from the IN endpoint's FIFO buffer.
    - **periph.c**: C source file that contains sample implementation of the functions declared in the **periph.h** header file (discussed above).
    - **wdf_dscr.h**: Header file that contains sample device descriptor information for the PIC18F4550 board.
    - **wdf_dscr.c**: Source file that contains definition of device descriptor data structures for the PIC18F4550 board.
    - **build.bat**: A utility for building the sample firmware code.
      Note: The build utility uses the evaluation firmware library (**wdf_microchip_18f4550_eval.lib**).
    - **loopback_eval.hex**: Sample loopback firmware for the PIC18F4550 board, created by building the sample code with the **build.bat** utility. **Note:** The firmware uses the evaluation firmware library (**wdf_microchip_18f4550_eval.lib**).
    - **loopback_eval.lkr**: A linker file for the loopback sample.
  - **msd**\ directory: Mass storage device sample. The sample implements a mass storage device for the Microchip PICTail daughter board with a Secure Digital Card (**SD Card**), using the PIC18F4550 mass storage evaluation library (**wdf_microchip_msd_18f4550_eval.lib**).

    The sample supports the following SD Cards: EP Memory 512MB; Lexar 256MB, 512MB and 1GB; SunDisk 128MB, 512MB and 2GB; SimpleTech 256MB and 1GB; Viking 512MB and 256MB; ATP 1GB.

* **periph.c**: C source file that contains sample implementation of the functions declared in the **periph.h** header file (discussed above).
* **wdf_dscr.h**: Header file that contains sample device descriptor information for the PIC18F4550 board.
* **wdf_dscr.c**: Source file that contains definition of device descriptor data structures for the PIC18F4550 board.
* **sdcard.h**: Header file that contains types and general definitions for the supported SD Cards (see list above).
* **sdcard.c**: Source file that implements the Microchip PIC18F4550 mass storage library's storage media access functions, declared in the **wdf_disk.h** header file [12.3.2], for the supported SD Cards.
* **build.bat**: A utility for building the sample firmware code. **Note:** The build utility uses the mass storage evaluation firmware library (**wdf_microchip_msd_18f4550_eval.lib**).
* **msd_eval.hex**: Sample mass storage firmware for the PIC18F4550 board, created by building the sample code with the **build.bat** utility. **Note:** The firmware uses the mass storage evaluation firmware library (**wdf_microchip_msd_18f4550_eval.lib**).
* **msd_eval.lkr**: A linker file for the mass storage sample.

12.3.3 The philips Directory

The **WinDriver\wdf\philips** directory contains the following directories:

- **d12**\ directory: Contains files for devices based on the PDIUSBD12.

The **d12**\ directory contains the following sub-directories and files:

- **include**\ directory:
  - **d12_lib.h**: Header file that contains firmware library types, general definitions and function prototypes for devices based on the PDIUSBD12. This file provides the interface of the PDIUSBD12's firmware library (**d12_eval.lib** – for evaluation users; For registered users the library's source code is created as part of the DriverWizard device firmware code generation – see explanation regarding the WinDriver USB device firmware libraries in section 12.3.5).
  - **types.h**: Header file that defines data types for the PDIUSBD12.

- **d12_io.h**: Header file that contains general firmware library definitions and functions declarations that are hardware-specific. This header provides the interface of the library's hardware abstraction layer.
  The default implementation of this file is targeted at the D12-ISA (PC) Eval Kit, version 1.4, which supports connection of a PDIUSBD12-based board to an x86 PC using an ISA card. However, registered users of WinDriver USB Device can modify the implementation of the library's hardware-specific APIs in this file and in **d12_io.c** [12.4.3.1], in order to support any other microcontroller (see also notes in section 12.4.3.2).
- **periph.h**: Header file that contains function prototypes for supporting USB peripheral device functionality for devices based on the PDIUSBD12. The functions' implementation is dependent on the specific configuration defined for the device. The **periph.c** source file that contains the implementation for your device is created by the DriverWizard when generating device firmware code, based on the USB device configuration that you define in the wizard see the description of the generated DriverWizard files [12.4.3.1].

• **lib**\ directory:

- **d12_eval.lib**: Evaluation firmware library for the PDIUSBD12 (see explanation below [12.3.5]).

• **samples**\ directory: Device firmware samples for the PDIUSBD12:

- **loopback**\ directory: Loopback sample: The sample implements a loopback, which fills the OUT endpoint's FIFO buffer with the data that is read from the IN endpoint's FIFO buffer.
  * **periph.c**: C source file that contains sample implementation of the functions declared in the **periph.h** header file (discussed above).
  * **wdf_dscr.c**: Source file that contains definition of device descriptor data structures for the PDIUSBD12.
  * **build.bat**: A utility for building the sample firmware code. **Note:** The build utility uses the evaluation firmware library (**d12_eval.lib**).
  * **LOOPBACK.EXE**: Sample loopback firmware for the PDIUSBD12, created by building the sample code with the **build.bat** utility. **Note:** The firmware uses the evaluation firmware library (**d12_eval.lib**).
  * **lb_eval.mak**: Makefile for building the loopback sample using the Turbo C compiler.

– **dma**\ directory: DMA sample: The sample demonstrates the PDIUSBD12's DMA capability by initiating a DMA transfer upon arrival of a relevant vendor request. DMA is performed using the x86 on-board DMA controller. In order to use a different controller, the hardware-dependent portions of the code need to be replaced with the hardware-specific code for the selected controller. Note that in order to modify the hardware-specific portions of the firmware library, you need to be a registered user of the WinDriver USB Device tool-kit (see section 12.3.5 for details).

  * **periph.c**: C source file that contains sample implementation of the functions declared in the **periph.h** header file (discussed above).
  * **wdf_dscr.c**: Source file that contains definition of device descriptor data structures for the PDIUSBD12.
  * **build.bat**: A utility for building the sample firmware code. **Note:** The build utility uses the evaluation firmware library (**d12_eval.lib**).
  * **D12DMA.EXE**: Sample DMA firmware for the PDIUSBD12, created by building the sample code with the **build.bat** utility. **Note:** The firmware uses the evaluation firmware library (**d12_eval.lib**).
  * **dma_eval.mak**: Makefile for building the DMA sample using the Turbo C compiler.

12.3.4 The silabs Directory

The **WinDriver****wdf****silabs** directory contains the following directories:

- **F320**\ directory: Contains files for devices based on the C8051F320 development board.
- **F340**\ directory: Contains a **readme.txt** file, which directs the user to the files in the **F320**\ directory, since the C8051F320 code can also be used to develop firmware for the C8051F340 board.

The **F320**\ directory contains the following sub-directories and files:

- **include**\ directory:
  – **wdf_silabs_lib.h**: Header file that contains firmware library types and function prototypes for devices based on the C8051F320 board. This file provides the interface of the board's firmware library (**wdf_silabs_f320_eval.lib** – for evaluation users; For registered users the library's source code is created as part of the DriverWizard device firmware code generation – see explanation regarding the WinDriver USB device firmware libraries in section 12.3.5).

  - **c8051f320.h**: Header file that contains general firmware library definitions for the C8051F320 board.
  - **c8051f320regs.h**: Header file that contains register/bits definitions for the C8051F320 board.
  - **periph.h**: Header file that contains function prototypes for supporting USB peripheral device functionality for devices based on the C8051F320 board. The functions' implementation is dependent on the specific configuration defined for the device. The **periph.c** source file that contains the implementation for your device is created by DriverWizard when generating device firmware code, based on the USB device configuration that you define in the wizard see the description of the generated DriverWizard files [12.4.3.1].

- **lib**\ directory:
  - **wdf_silabs_f320_eval.lib**: Evaluation firmware library for the C8051F320 board (see explanation below [12.3.5]).
- **samples**\ directory: Device firmware samples for the C8051F320 board:
  - **loopback**\ directory: Loopback sample: The sample implements a loopback, which fills the OUT endpoint's FIFO buffer with the data that is read from the IN endpoint's FIFO buffer.
    * **periph.c**: C source file that contains sample implementation of the functions declared in the **periph.h** header file (discussed above).
    * **wdf_dscr.h**: Header file that contains sample device descriptor information for the C8051F320 board.
    * **wdf_dscr.c**: Source file that contains definition of device descriptor data structures for the C8051F320 board.
    * **build.bat**: A utility for building the sample firmware code.
      Note: The build utility uses the evaluation firmware library (**wdf_silabs_f320_eval.lib**).
    * **loopback_eval.hex**: Sample loopback firmware for the C8051F320 board, created by building the sample code with the **build.bat** utility. **Note:** The firmware uses the evaluation firmware library (**wdf_silabs_f320_eval.lib**).

12.3.5 The WinDriver USB Device Firmware Libraries

When generating firmware code with DriverWizard using the registered version of the WinDriver USB Device tool-kit, the generated code includes WinDriver USB Device firmware library source files, which contain API for performing common USB firmware functionality.

When selecting to generate mass storage firmware code for the **PIC18F4550** board using the registered version of the tool-kit, the generated code also includes the source files for the **wdf_microchip_msd_18f4550_eval.lib** mass storage firmware library, which contain API for development of mass storage USB devices, in accordance with the USB Mass Storage Class Specification.

For details regarding the generated DriverWizard files, refer to section 12.4.3.1.

The library source files are not part of the evaluation version of the tool-kit. In order to enable an evaluation of WinDriver USB Device, this tool-kit includes pre-compiled evaluation libraries, which are utilized by the device firmware samples and the generated DriverWizard evaluation firmware code.

The evaluation libraries provide the same functionality as the registered library files, subject to the following single limitation: they only enable you to perform a pre-set number of transfers (25,000 – for the standard evaluation libraries / 1,000,000 – for the PIC18F4550 mass storage evaluation library). When this amount is exceeded the library will cease to work.

For information on how to register firmware developed during the evaluation period, after receiving a WinDriver USB Device license, refer to the notes in section 12.4.3.2.

12.3.6 Building the Sample Code

To build any **WinDriver\wdf\<vendor>\<hardware>\samples\<sample_name>** sample (e.g. the Cypress loopback sample – **WinDriver\wdf\cypress\FX2LP\samples\loopback**), use the **build.bat** utility for the selected sample (e.g. **FX2LP\samples\loopback\build.bat**).

Before running the utility, verify that the information in the **build.bat** file matches your system configuration:

- For the **Cypress EZ-USB FX2LP CY7C68013A** and **Silicon Laboratories C8051F320 and C8051F340** boards: verify that the `KEIL` variable in the **build.bat** file is set to point to the location of your Keil development tools directory. The default Keil directory used in the **build.bat** files is **C:\Keil**. If you installed Keil in a different location, modify the following line in the **build.bat** file in order to point to the correct location:

  ```
  set KEIL=C:\Keil
  ```

For example, if you installed Keil under **D:\MyTools\Keil**, modify the line to:

```
set KEIL=D:\MyTools\Keil
```

- For the **Cypress EZ-USB FX2LP CY7C68013A** samples, verify that the `CYPRESS` variable in the **build.bat** file is set to point to the location of your Cypress EZ-USB Development Kit. The default directory used in the **build.bat** file is **C:\Cypress**. If you installed the Cypress EZ-USB Development Kit in a different location, modify the following line in the **build.bat** file in order to point to the correct location:

  ```
  set CYPRESS=C:\Cypress
  ```

  For example, if you installed the Cypress EZ-USB Development Kit under **D:\Cypress**, modify the line to:

  ```
  set CYPRESS=D:\Cypress
  ```

- For the **Microchip PIC18F4550** samples, verify that the `MCC` variable in the **build.bat** file is set to point to the location of your MCC18 directory. The default directory used in the **build.bat** file is **C:\mcc18**. If you installed the MCC18 Toolchain in a different location, modify the following line in the **build.bat** file in order to point to the correct location:

  ```
  set MCC=C:\mcc18
  ```

  For example, if you installed the MCC18 Toolchain under **D:\microchip\mcc18**, modify the line to:

  ```
  set MCC=D:\microchip\mcc18
  ```

- For the **Philips PDIUSBD12** samples, verify that the `TURBOC` variable in the **build.bat** file is set to point to the location of your Turbo C directory. The default directory used in the **build.bat** file is **c:\borlandc**. If you installed the Turbo C compiler in a different location, modify the following line in the **build.bat** file in order to point to the correct location:

  ```
  set TURBOC=c:\borlandc\
  ```

  For example, if you installed the Turbo C compiler under **D:\TurboC**, modify the line to:

  ```
  set TURBOC=d:\TurboC\
  ```

- Run the **build.bat** utility to build the sample firmware.

12.4 WinDriver USB Device Development Process

Use WinDriver USB Device to develop firmware for your USB device (based on any of the target hardware) by following the steps below:

12.4.1 Define the Device USB Interface

Use the WinDriver USB Device DriverWizard utility to define your device's USB interface:

1. Run DriverWizard, using either of the following methods:
   - Click **Start | Programs | WinDriver | DriverWizard**
   - Double-click the DriverWizard icon on your desktop
   - Run **WinDriver****wizard****wdwizard.exe**, either by double-clicking the executable file or by running it from a command-line prompt.
2. Check the **New device firmware project** option in the wizard's **Choose Your Project** dialogue and click **Next »**. Alternatively, you can also select to create a new device firmware project from DriverWizard's **File** menu or by clicking the firmware project icon in the wizard's toolbar.

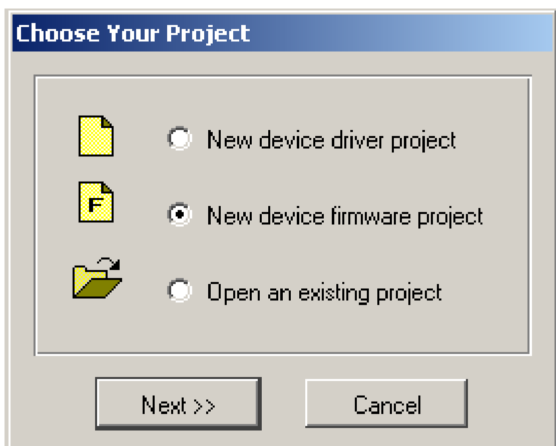


Figure 12.1: Create Device Firmware Project

3. Select your target hardware from the **Choose Your Development Board** dialogue and click **OK**.

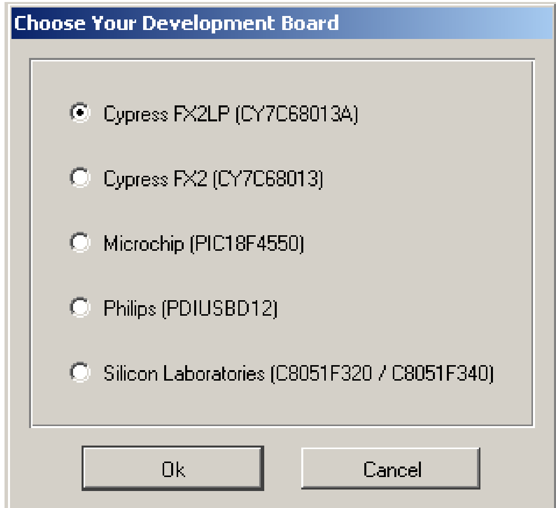


Figure 12.2: Choose Your Development Board

4. If you selected the Microchip PIC18F4550 board, select the desired device functionality from the **Choose Your Device Function** dialogue.

Figure 12.3: Microchip – Choose Your Device Function

5. In the **Edit Device Descriptor** dialogue, define the basic device descriptor information for your target device – i.e. the vendor and device IDs, manufacturer and device descriptions, device class and sub-class, etc.

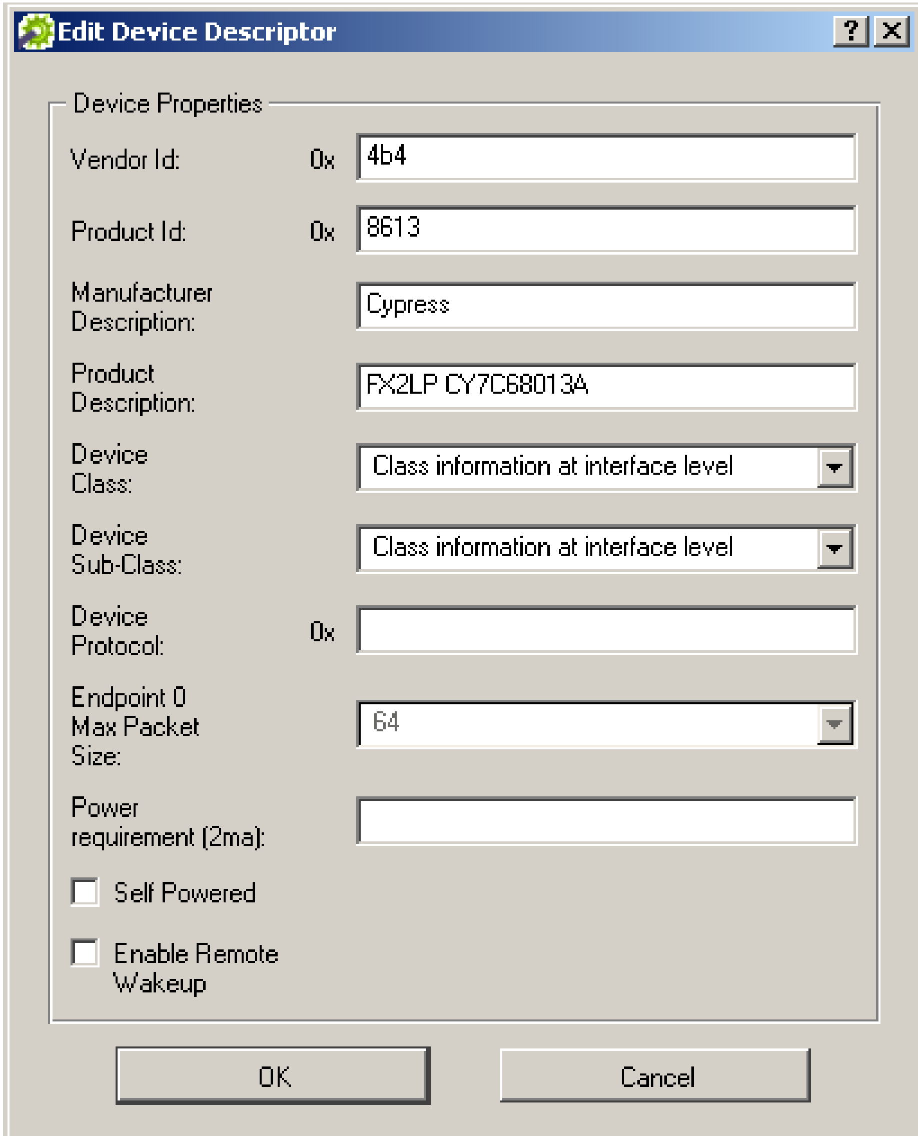


Figure 12.4: Edit Device Descriptor

6. In the **Configure Your Device** dialogue, proceed to define the desired USB configuration for your device.

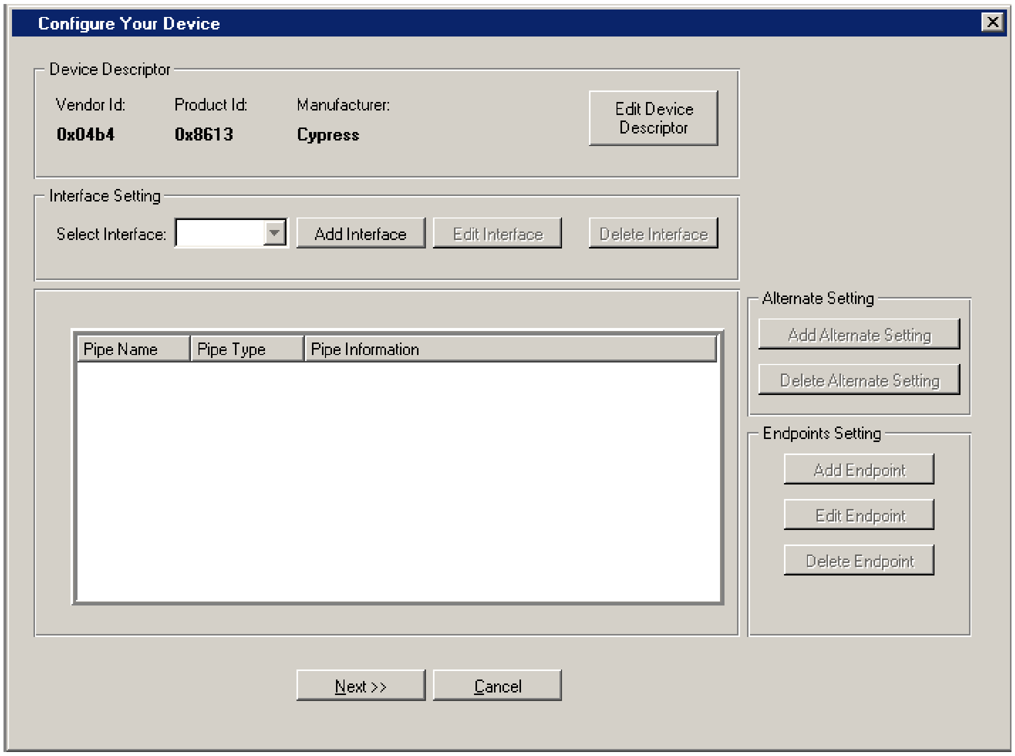


Figure 12.5: Configure Your Device

The dialogue enables you to add device interfaces, add alternate settings for each interface, and add the required endpoints for each alternate setting.

NOTES

- If you select to define more than one interface, DriverWizard will generate firmware code for a **composite device**. The wizard will warn you about this when you select to add a second interface.
- Definition of multiple interfaces is not currently supported for the Silicon Laboratories C8051F320 and C8051F340 and Philips PDIUSBD12 hardware.
- When generating **mass storage** firmware for the Microchip PIC18F4550 board there is no option to add interfaces or alternate settings, since such firmware is defined for a specific interface and alternate setting.

When adding components, the wizard allows you define the relevant attributes for each component (such as the interface's class and sub-class or the endpoint's address, transfer type, maximum packet size, etc.). The wizard further assists you by only providing the relevant configuration options for your device and by warning you if there is a potential error in your configuration definitions.

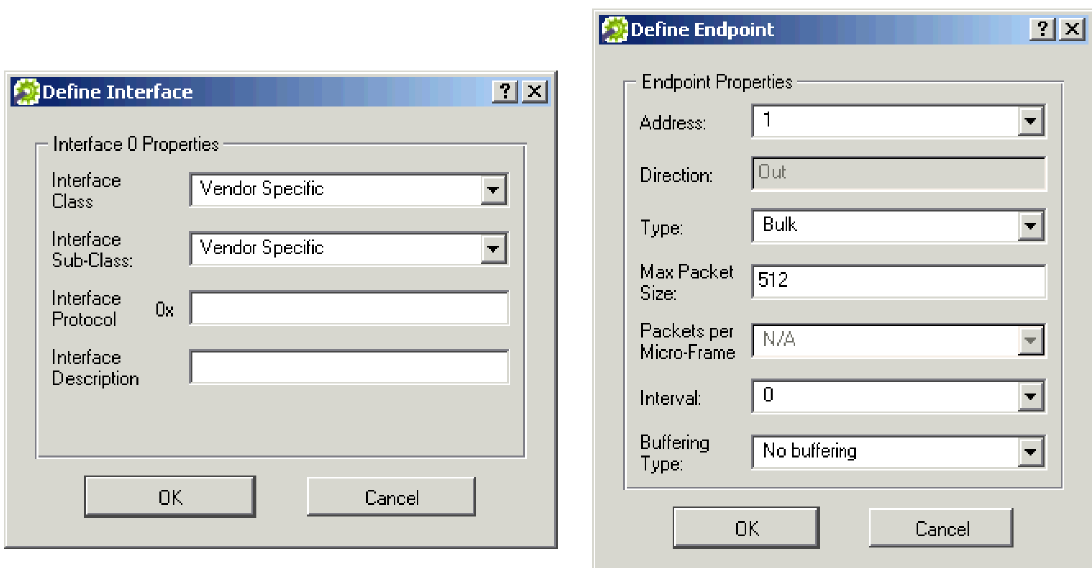


Figure 12.6: Define Interfaces and Endpoints

When generating **Philips PDIUSBD12** firmware, the wizard simplifies the configuration of the main endpoints, by allowing you to select the main isochronous endpoints (if any) and defining these endpoints for you.
Figure 12.7 demonstrates definition of the main endpoints by the wizard when selecting the `ISO_IN_OUT` mode, which defines both of the main endpoints as isochronous.

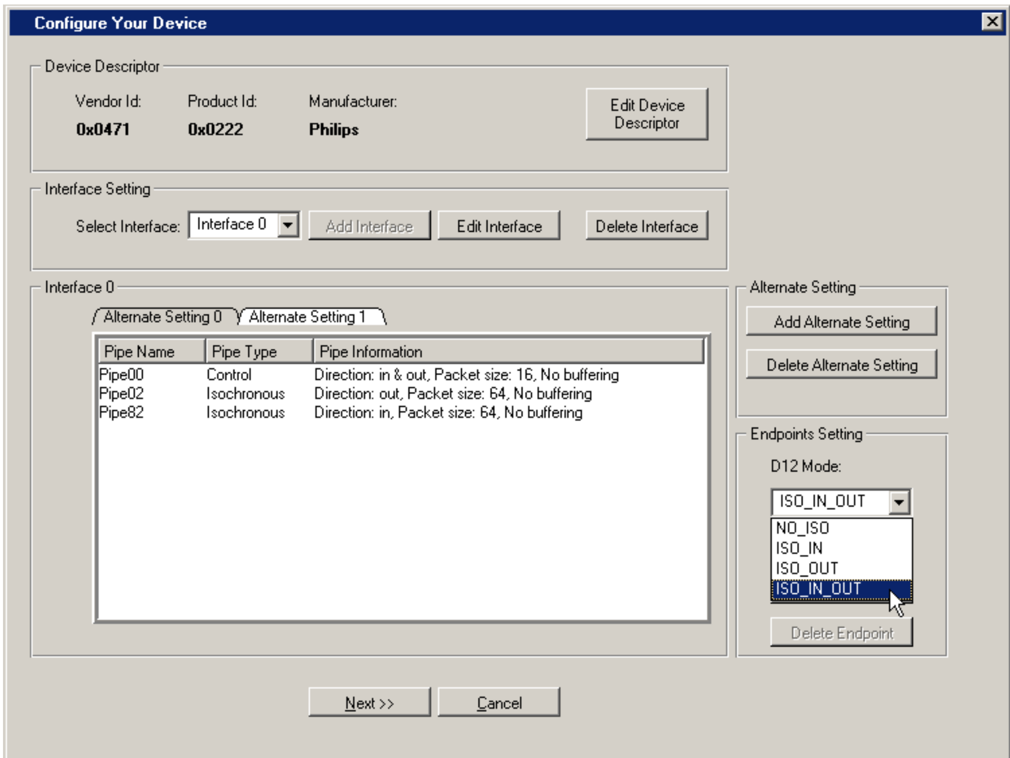


Figure 12.7: Philips PDIUSBD12 – Define Main Endpoint Pipes

More information on how to configure the endpoints on the **Cypress EZ-USB FX2LP CY7C68013A** development board can be found at the end of this section [12.4.1.1].

When defining the configuration for a **mass storage** device based on the **Microchip PIC18F4550** board, you can select the **Edit Inquiry Descriptor** option to define how the device should reply to a SCSI Inquiry request from the host.

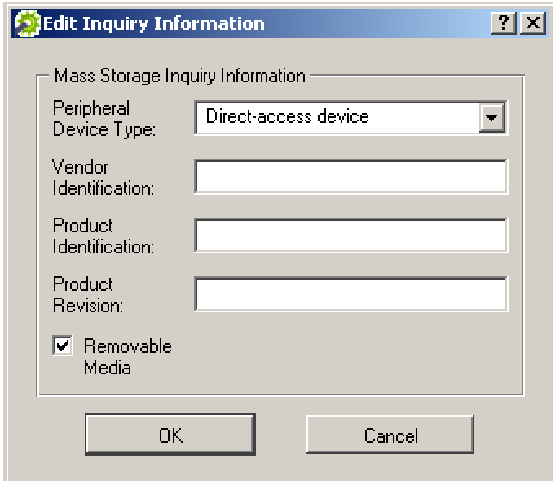


Figure 12.8: Microchip PIC18F4550 Mass Storage – Edit Inquiry Info

You can delete any component that you have added or edit the configuration information, at any time, from the device configuration dialogue.

7. You can select to save your DriverWizard device firmware project at any stage, either from the **File** menu or using the relevant icon in the wizard's toolbar. This will enable you to open the saved **xxx.wdp** device firmware project from DriverWizard at a later time and resume where you left off.

When you have finished defining the device's USB interface, proceed to generate device firmware code, based on your DriverWizard definitions, as outlined in the following section [12.4.2].

12.4.1.1 EZ-USB Endpoint Buffers Configuration

*This section contains a quote from section 1.18 of the EZ-USB Technical Reference Manual (**EZ-USB_TRM.pdf**) regarding EZ-USB endpoint buffers configuration. This information can be useful when using DriverWizard to define the endpoint configuration for devices based on the **Cypress EZ-USB FX2LP CY7C68013A** development board.*
*For more information, refer to the EZ-USB Technical Manual, which is available under the **Cypress****USB****Doc****FX2LP** directory or on-line at:* `http://www.keil.com/dd/docs/datashts/cypress/fx2_trm.pdf`.

The USB 2.0 Specification defines an endpoint as a source or sink of data. Since USB is a serial bus, a device endpoint is actually a FIFO which sequentially empties or fills with USB data bytes. The host selects a device endpoint by sending a 4-bit address and a direction bit. Therefore, USB can uniquely address 32 endpoints, IN0 through IN15 and OUT0 through OUT15.

From the EZ-USB's point of view, an endpoint is a buffer full of bytes received or held for transmission over the bus. The EZ-USB reads host data from an OUT endpoint buffer, and writes data for transmission to the host to an IN endpoint buffer.

EZ-USB contains three 64-byte endpoint buffers, plus 4 KB of buffer space that can be configured 12 ways, as indicated in Figure 1-16. The three 64-byte buffers are common to all configurations.

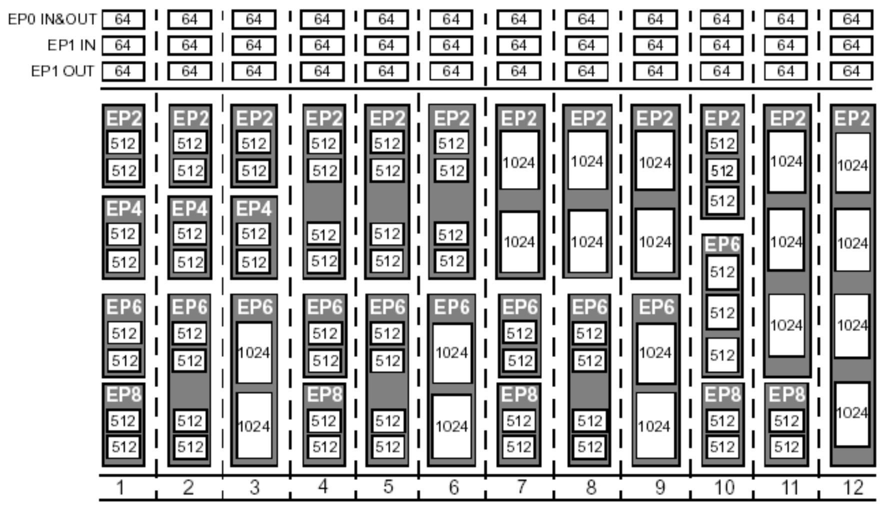


Figure 12.9: EZ-USB Endpoint Buffers

12.4.2 Generate Device Firmware Code

Generate device firmware code from the **Configure Your Device** dialogue, using either of the following methods:

- Click the **Next »** button or use the Alt+N short-cut key.
- Select the **Generate Code** toolbar icon.
- From the **Build** menu select the **Generate Code** option.

The wizard's **Select Code Generation Options** dialogue will be displayed:

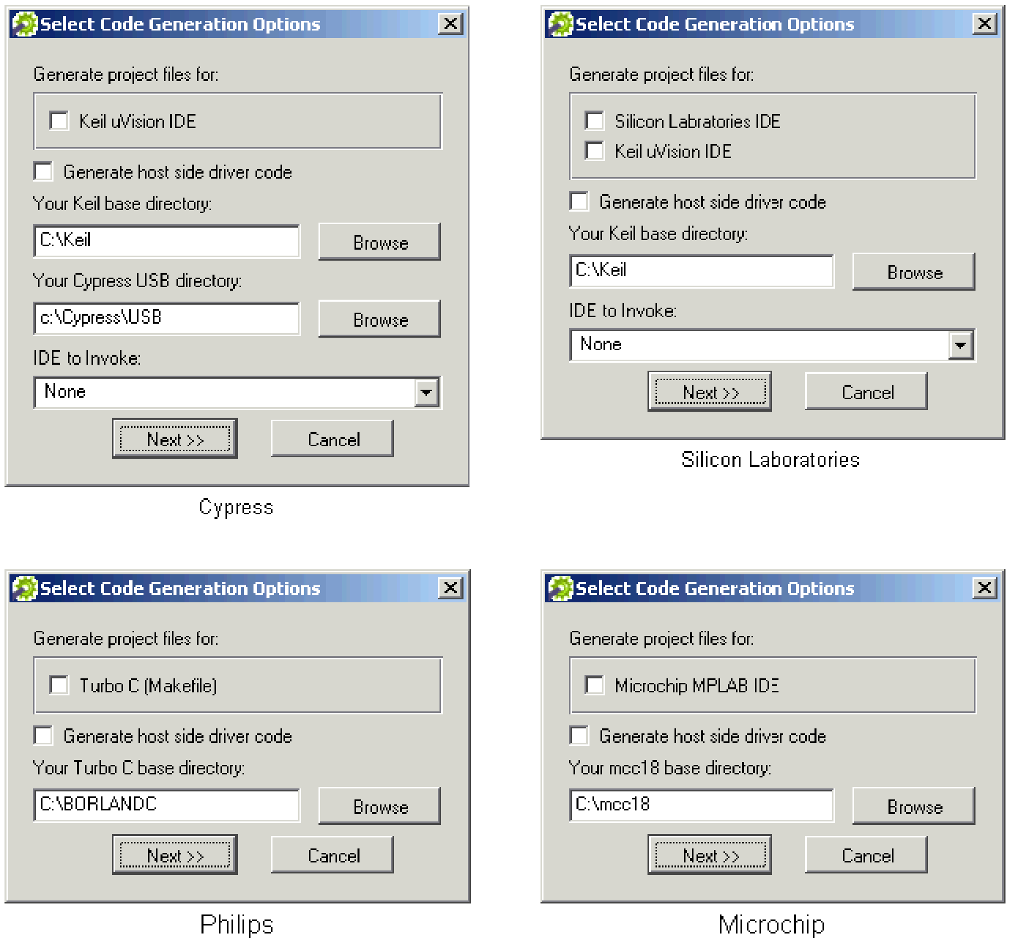


Figure 12.10: Firmware Code Generation

Verify that all directory paths in the device firmware code generation dialogue point to the correct locations on your PC:

- For the **Cypress EZ-USB FX2LP CY7C68013A** and **Silicon Laboratories C8051F320 and C8051F340** hardware, **Your Keil base directory** should point to the installation directory of the Keil Cx51 development tools for 8x51.

- For the **Cypress EZ-USB FX2LP CY7C68013A** hardware, **Your Cypress USB directory** should point to the location of the Cypress EZ-USB FX2LP Development Kit **Cypress\USB** directory.
- For the **Microchip PIC18F4550** hardware, **Your mcc18 base directory** should point to the installation directory of the MCC18 Toolchain.
- For the **Philips PDIUSBD12** hardware, **Your Turbo C base directory** should point to the installation directory of the Turbo C compiler.

You can select to generate a specific **project/make file** for any of the supported development environments for your target hardware [12.2] by checking the relevant check-box in the **Select Code Generation Options** dialogue.

When selecting to generate a project file for the **Keil μVision IDE** or **Silicon Laboratories IDE**, the wizard will automatically change the **IDE to Invoke** to your selected IDE. If you do not change the IDE to invoke, the wizard will attempt to invoke this IDE after generating the code.

The **Generate host side driver code** option, shown in the dialogue screen shots above, is available during the evaluation of the tool-kit and for registered users of the WinDriver USB Driver Development Kit. When this option is selected, in addition to the device firmware code the wizard will also generate a skeletal WinDriver USB device driver application for your USB device (as defined in the wizard). – see Chapter 5 and section 12.4.5 for details regarding the DriverWizard device driver code generation.

12.4.3 Develop the Device Firmware

After you have generated the firmware code with the wizard, you are free to modify it, as needed, in order to implement your desired firmware functionality, using the library and generated WinDriver USB Device firmware API to facilitate your development efforts.

The API of the USB firmware libraries and generated code is described in detail in Appendices B – E.

NOTE
When modifying the WinDriver library and generated device firmware code, make sure that your code complies with your target hardware's specification:

- For the **FX2LP CY7C68013A** board: **EZ-USB_TRM.pdf** – see specifically section 15.6 *Endpoint Configuration*.
  This document is available under the**Cypress\USB\Doc\FX2LP** directory or on-line at: `http://www.keil.com/dd/docs/datashts/cypress/fx2_trm.pdf`.
- For the **PIC18F4550** board: **39632b.pdf** – see specifically section 17.3 *USB RAM* and 17.4 *Buffer Descriptors and the Buffer Descriptors Table*. This document is available on-line at:
  `http://ww1.microchip.com/downloads/en/DeviceDoc/39632b.pdf`.
- For the **PDIUSBD12**: **PDIUSBD12-08.pdf**, available on-line from:
  `http://www.semiconductors.philips.com/pip/PDIUSBD12D.html#datasheet`.
- For the **Silicon Laboratories C8051F320 and C8051F340** boards:
  C8051F32xRev1_1.pdf (C8051F320) / **C8051F34xRev0_5.pdf** (C8051F340) – see specifically the *FIFO Management* section (C8051F320 – 15.5 / C8051F340 – 16.5) and the *Configuring Endpoints 1-3* section (C8051F320 – 15.11 / C8051F340 – 16.11). These documents are available under the **Silabs\MCU\Documentation\Datasheets** directory (if you installed the Silicon Laboratories IDE) or on-line at:
  `http://www.keil.com/dd/docs/datashts/silabs/c8051f32x.pdf` (C8051F320)
  `http://www.keil.com/dd/docs/datashts/silabs/c8051f34x.pdf` (C8051F340)

12.4.3.1 The Generated DriverWizard USB Device Firmware Files

When generating device firmware code, DriverWizard creates an **xxx_FW** directory, which contains the following files:

- **periph.c**: C source file, which includes implementation of functions for supporting USB peripheral device functionality for your device, declared in the **WinDriver\wdf\<vendor> \<hardware>\include\periph.h** header file (e.g. **WinDriver\wdf\cypress\FX2LP\include\periph.h**) [12.3.1]. The functions' implementation is derived from the specific device configuration that you defined with DriverWizard.
- Device descriptor information, which utilizes the device descriptor information that you defined with DriverWizard:
  - For the **Cypress EZ-USB FX2LP CY7C68013A** hardware:
    wdf_dscr.a51: Assembly file.

  - For the **Microchip PIC18F4550**, **Philips PDIUSBD12** and **Silicon Laboratories C8051F320 and C8051F340** hardware: **wdf_dscr.c** and **wdf_dscr.h** (Microchip and Silicon Laboratories): C files.

- **build.bat**: A command-line utility for building the firmware code.
- **xxx.Uv2/mcp/wsp/mak**: Project/make file for building the code using your selected compiler/IDE (Keil μVision / Microchip MPLAB / Silicon Laboratories IDE / Turbo C), provided you selected the relevant compiler/IDE from the **Select Code Generation Options** dialogue.
- For the **Microchip PIC18F4550** board:
  - **xxx.lkr**: A linker file.
  - When generating **mass storage** device firmware for the board:
    * **wdf_xxx_hw.c**: C source file, which contains stubs for implementation of the hardware-dependent storage media access functions, declared in the header file **18F4550\include\class\msd\wdf_disk.h** [12.3.2].

The following files contain the source code of the WinDriver USB Device firmware library. These files are generated only when using the **registered version** of the WinDriver USB Device tool-kit (see notes in section 12.4.3.2 regarding the differences between the evaluation and registered version):

- **main.c**: C source file, which contains the implementation of the firmware's main entry point. For devices based on the Silicon Laboratories C8051F320 and C8051F340 and Philips PDIUSBD12 hardware, the file also implements the required USB interrupt service routine – `USB_ISR()` (Silicon Laboratories) / `UsbISR()` (Philips). Note: The implementation of the ISR in the Philips code is platform dependent. The default implementation is for the D12-ISA (PC) Eval Kit and therefore x86-targeted, however you can modify this implementation to support any other appropriate microcontroller (see also notes in section 12.4.3.2).
- **wdf_<vendor>_lib.c** – for Cypress, Microchip and Silicon Laboratories (e.g. **wdf_cypress_lib.c**) / **<hardware>_lib.c** – for Philips (e.g. **d12_lib.c**): C source file, which contains the implementation of the WinDriver USB Device firmware library functions for the selected target hardware.
- For the **Microchip PIC18F4550** board:
  - **wdf_usb9.c**: C source file, which contains the implementation of the firmware library Chapter 9 USB descriptors functions, declared in the **wdusb9_.h** header [12.3.2].

- When generating **mass storage** device firmware for the board:
  * **wdf_msd.c**: C source file, which contains the implementation of the mass storage device class firmware functions, declared in the **WinDriver\wdf\microchip\18F4550\include\class\msd\wdf_msd.h** header file [12.3.2].

- For the **Philips PDIUSBD12**:
  - **d12_ci.h** and **d12_ci.c**: Header file and C source file (respectively), which contain the PDIUSBD12 command interface general definitions and function declarations (**d12_ci.h**) and the implementation of the command functions (**d12_ci.c**).
  - **d12_io.c**: C source file, which contains the implementation of the hardware-specific firmware library functions, declared in the **d12_io.h** header file [12.3.3].

12.4.3.2 Build the Generated DriverWizard Firmware

To build the generated firmware code for your device, use any of the following alternative methods:

- Run the generated **build.bat** utility from a command-line prompt.
- If you selected to generate a project file for one of the supported IDEs (Keil μVision – for the Cypress and Silicon Laboratories boards; Microchip MPLAB – for Microchip; Silicon Laboratories IDE – for Silicon Laboratories), you can simply build the generated project using your selected IDE.

The build output is an **xxx.hex** firmware file – for the Cypress, Microchip and Silicon Laboratories boards / **XXX.EXE** firmware file – for the Philips PDIUSBD12 (where **xxx** is the name you selected for your firmware project).

NOTES

- The generated **build.bat** and compiler-specific project/make files are different for the registered and for the evaluation version of WinDriver USB Device and produce different output. The **evaluation version** of these files uses the evaluation firmware libraries and the created firmware will be limited to a maximum of 25,000 transfers – for the standard libraries / 1,000,000 transfers – for the Microchip PIC18F4550 mass storage library (see above [12.3.5]). The **registered version** uses the generated library source files and is not subject to the evaluation limitations.
- The **Philips PDIUSBD12** firmware library and the samples, which utilize this library, are targeted at the D12-ISA (PC) Eval Kit, which supports connection of a PDIUSBD12-based board to an x86 PC using an ISA card. The created firmware is therefore in the form of a DOS executable file. However, **registered** WinDriver USB Device users can modify the hardware-specific x86 portions of the library (specifically in **d12_io.h**, **d12_io.c** [12.4.3.1] and the ISR in **main.c**) and samples source code in order to support any other appropriate microcontroller, and then build and download the firmware using the supported method for the selected microcontroller.
- To **register your evaluation firmware**, after registering your WinDriver USB Device tool-kit, open the DriverWizard device firmware project file that you created during the evaluation period (**xxx.wdp**) and re-generate the firmware code with the wizard in order to create new registered versions of the **build.bat** and project files. Then use these files to build a registered, full-featured, firmware and download the firmware to the device.

12.4.3.3 Download the Firmware to the Device

After building the firmware, download it to the hardware using the hardware vendor's firmware download tools.

NOTE: The **Cypress EZ-USB FX2LP CY7C68013A** firmware and the standard **Microchip PIC18F4550** firmware can also be downloaded using the WinDriver USB Driver Development Kit's sample firmware download application (Cypress: see **download_sample.exe** in the **WinDriver\cypress\firmware_sample\WIN32** directory; Microchip: see **bootloader_demo.exe** in the **WinDriver\microchip\pic18f4550\bootloader_sample\WIN32** directory). This option is available for registered or evaluation users of the WinDriver USB Driver Development Kit.

12.4.4 Diagnose and Debug Your Hardware

Once you have downloaded the firmware to the device, you can use the DriverWizard utility to debug the firmware, as outlined in section 5.2 (refer to the USB explanations in this Chapter). **NOTE:** The device driver code generation option described in section 5.2 is not part of the WinDriver USB Device license.

12.4.5 Develop a USB Device Driver

When the device development is completed, if you have also purchased a license for the WinDriver USB Driver Development Kit, or if you are using the evaluation version of WinDriver, you can proceed to use WinDriver to develop a driver for your device, as explained in Chapter 6.
As indicated in section 12.4.2 above, if you have a compatible license you will also be given the option to generate a skeletal WinDriver USB device driver application from DriverWizard's firmware generation dialogue.

Appendix A

WinDriver USB PC Host API Reference

NOTE
This function reference is C oriented. The WinDriver .NET, Visual Basic and Delphi APIs have been implemented as closely as possible to the C APIs, therefore .NET, VB and Delphi programmers can also use this reference to better understand the WinDriver APIs for their selected development language. For the exact API implementation and usage examples for your selected language, refer to the WinDriver .NET/VB/Delphi source code.

A.1 WinDriver USB (WDU) Library Overview

This section provides a general overview of WinDriver's USB Library (WDU), including:

- An outline of the `WDU_xxx` API calling sequence – see section A.1.1.
- Instructions for upgrading code developed with the previous WinDriver USB API, used in version 5.22 and earlier, to use the improved `WDU_xxx` API – see section A.1.2.
  If you do not need to upgrade USB driver code developed with an older version of WinDriver, simply skip this section.

The WDU library's interface is found in the **WinDriver/include/wdu_lib.h** and **WinDriver/include/windrvr.h** header files, which should be included from any source file that calls the WDU API. (**wdu_lib.h** already includes **windrvr.h**).

A.1.1 Calling Sequence for WinDriver USB

The WinDriver WDU_xxx USB API is designed to support event-driven transfers between your user-mode USB application and USB devices. This is in contrast to earlier versions, in which USB devices were initialized and controlled using a specific sequence of function calls.

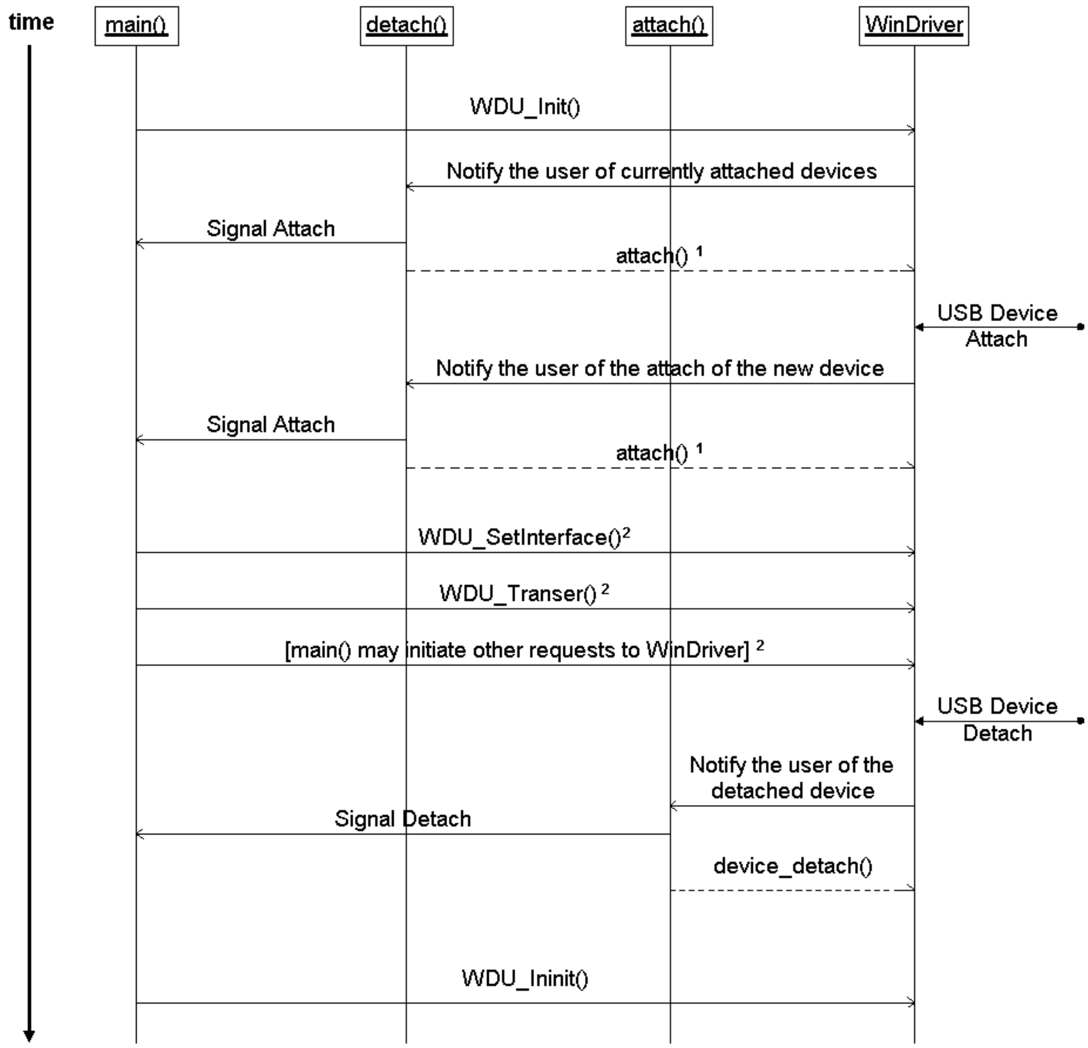
You can implement the three user callback functions specified in the next section: WDU_ATTACH_CALLBACK [A.2.1], WDU_DETACH_CALLBACK [A.2.2] and WDU_POWER_CHANGE_CALLBACK [A.2.3] (at the very least WDU_ATTACH_CALLBACK). These functions are used to notify your application when a relevant system event occurs, such as the attaching or detaching of a USB device. For best performance, minimal processing should be done in these functions.

Your application calls WDU_Init() [A.3.1] and provides the criteria according to which the system identifies a device as relevant or irrelevant. The WDU_Init() function must also pass pointers to the user callback functions.

Your application then simply waits to receive a notification of an event. Upon receipt of such a notification, processing continues. Your application may make use of any functions defined in the high- or low-level APIs below. The high-level functions, provided for your convenience, make use of the low-level functions, which in turn use IOCTLs to enable communication between the WinDriver kernel module and your user-mode application.

When exiting, your application calls WDU_Uninit() [A.3.6] to stop listening to devices matching the given criteria and to un-register the notification callbacks for these devices.

The following figure depicts the calling sequence described above. Each vertical line represents a function or process. Each horizontal arrow represents a signal or request, drawn from the initiator to the recipient. Time progresses from top to bottom.



[1] If the WD_ACKNOWLEDGE flag was set in the call to WDU_Init(), the attach() callback should return TRUE to accept control of the device or FALSE otherwise.

[2] Only possible if the attach() callback returned TRUE.

Figure A.1: WinDriver USB Calling Sequence

The following piece of meta-code can serve as a framework for your user-mode application's code:

```
attach()
{
    ...
    if this is my device
        /*
        Set the desired alternate setting ;
        Signal main() about the attachment of this device
        */

        return TRUE;
    else
        return FALSE;
}

detach()
{
    ...
    signal main() about the detachment of this device
    ...
}

main()
{
    WDU_Init(...);

    ...
    while (...)
    {
        /* wait for new devices */

        ...

        /* issue transfers */

        ...
    }
    ...
    WDU_Uninit();
}
```

A.1.2 Upgrading from the WD_xxx USB API to the WDU_xxx API

The WinDriver WDU_xxx USB API, provided beginning with version 6.00, is designed to support event-driven transfers between your user-mode USB application and USB devices. This is in contrast to earlier versions, in which USB devices were initialized and controlled using a specific sequence of function calls.

As a result of this change, you will need to modify your USB applications that were designed to interface with earlier versions of WinDriver to ensure that they will work with WinDriver v6.X on all supported platforms and not only on Microsoft Windows. You will have to reorganize your application's code so that it conforms with the framework illustrated by the piece of meta-code provided in section A.1.1.

In addition, the functions that collectively define the USB API have been changed. The new functions, described in the next few sections, provide an improved interface between user-mode USB applications and the WinDriver kernel module. Note that the new functions receive their parameters directly, unlike the old functions, which received their parameters using a structure.

The table below lists the legacy functions in the left column and indicates in the right column which function or functions replace(s) each of the legacy functions. Use this table to quickly determine which new functions to use in your new code.

Problem	**Solution**
High Level API	
This function...	*has been replaced by...*
WD_Open() WD_Version() WD_UsbScanDevice()	WDU_Init() [A.3.1]
WD_UsbDeviceRegister()	WDU_SetInterface() [A.3.2]
WD_UsbGetConfiguration()	WDU_GetDeviceInfo() [A.3.4]
WD_UsbDeviceUnregister()	WDU_Uninit() [A.3.6]
Low Level API	
This function...	*has been replaced by...*
WD_UsbTransfer()	WDU_Transfer() [A.3.7] WDU_TransferDefaultPipe() [A.3.9] WDU_TransferBulk() [A.3.10] WDU_TransferIsoch() [A.3.11] WDU_TransferInterrupt() [A.3.12]
USB_TRANSFER_HALT option	WDU_HaltTransfer() [A.3.13]
WD_UsbResetPipe()	WDU_ResetPipe() [A.3.14]
WD_UsbResetDevice() WD_UsbResetDeviceEx()	WDU_ResetDevice() [A.3.15]

A.2 USB User Callback Functions

A.2.1 WDU_ATTACH_CALLBACK()

PURPOSE

• WinDriver calls this function when a new device, matching the given criteria, is attached, provided it is not yet controlled by another driver.
This callback is called once for each matching interface.

PROTOTYPE

```
typedef BOOL (DLLCALLCONV *WDU_ATTACH_CALLBACK)(
    WDU_DEVICE_HANDLE hDevice,
    WDU_DEVICE *pDeviceInfo,
    PVOID pUserData);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ pDeviceInfo	WDU_DEVICE*	Input
➢ pUserData	PVOID	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface
pDeviceInfo	Pointer to a USB device information structure [A.4.3]; Valid until the end of the function
pUserData	Pointer to user-mode data for the callback, as passed to `WDU_Init()` [A.3.1] within the event table parameter (`pEventTable->pUserData`)

RETURN VALUE

If the `WD_ACKNOWLEDGE` flag was set in the call to `WDU_Init()` [A.3.1] (within the `dwOptions` parameter), the callback function should check if it wants to control the device, and if so return `TRUE` (otherwise – return `FALSE`).
If the `WD_ACKNOWLEDGE` flag was not set in the call to `WDU_Init()`, then the return value of the callback function is insignificant.

A.2.2 WDU_DETACH_CALLBACK()

PURPOSE

• WinDriver calls this function when a controlled device has been detached from the system.

PROTOTYPE

```
typedef void (DLLCALLCONV *WDU_DETACH_CALLBACK) (
    WDU_DEVICE_HANDLE hDevice,
    PVOID pUserData);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ pUserData	PVOID	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface
pUserData	Pointer to user-mode data for the callback, as passed to `WDU_Init()` [A.3.1] within the event table parameter (`pEventTable->pUserData`)

RETURN VALUE

None

A.2.3 WDU_POWER_CHANGE_CALLBACK()

PURPOSE

• WinDriver calls this function when a controlled device has changed its power settings.

PROTOTYPE

```
typedef BOOL (DLLCALLCONV *WDU_POWER_CHANGE_CALLBACK)(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPowerState,
    PVOID pUserData);
```

PARAMETERS

Name	Type	Input/Output
➢ dwPowerState	DWORD	Input
➢ pUserData	PVOID	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface
dwPowerState	Number of the power state selected
pUserData	Pointer to user-mode data for the callback, as passed to `WDU_Init()` [A.3.1] within the event table parameter (`pEventTable->pUserData`)

RETURN VALUE

TRUE/FALSE. Currently there is no significance to the return value.

REMARKS

- This callback is supported only in Windows operating systems, starting from Windows 2000.

A.3 USB Functions

A.3.1 WDU_Init()

PURPOSE

• Starts listening to devices matching input criteria and registers notification callbacks for these devices.

PROTOTYPE

```
DWORD WDU_Init(
    WDU_DRIVER_HANDLE *phDriver,
    WDU_MATCH_TABLE *pMatchTables,
    DWORD dwNumMatchTables,
    WDU_EVENT_TABLE *pEventTable,
    const char *sLicense,
    DWORD dwOptions);
```

PARAMETERS

Name	Type	Input/Output
➢ phDriver	WDU_DRIVER_HANDLE *	Output
➢ pMatchTables	WDU_MATCH_TABLE*	Input
➢ dwNumMatchTables	DWORD	Input
➢ pEventTable	WDU_EVENT_TABLE*	Input
➢ sLicense	const char*	Input
➢ dwOptions	DWORD	Input

Description

Name	Description
phDriver	Handle to the registration of events & criteria
pMatchTables	Array of match tables [A.4.1] defining the devices' criteria
dwNumMatchTables	Number of elements in pMatchTables
pEventTable	Pointer to an event table structure [A.4.2], which holds the addresses of the user-mode device status change notification callback functions [A.2] and the data to pass to the callbacks
sLicense	WinDriver's license string
dwOptions	Can be zero or : • **`WD_ACKNOWLEDGE`** – the user can seize control over the device when returning value in `WDU_ATTACH_CALLBACK` [A.2.1]

Return Value

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.2 WDU_SetInterface()

PURPOSE

• Sets the alternate setting for the specified interface.

PROTOTYPE

```
DWORD WDU_SetInterface(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwInterfaceNum,
    DWORD dwAlternateSetting);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ dwInterfaceNum	DWORD	Input
➢ dwAlternateSetting	DWORD	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface
dwInterfaceNum	The interface's number
dwAlternateSetting	The desired alternate setting value

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.3 WDU_GetDeviceAddr()

PURPOSE

• Gets the USB address for a given device.

PROTOTYPE

```
DWORD WDU_GetDeviceAddr(
    WDU_DEVICE_HANDLE hDevice,
    ULONG *pAddress);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ pAddress	ULONG	Output

DESCRIPTION

Name	Description
hDevice	A unique identifier for a device/interface
pAddress	A pointer to the address number returned by the function

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

REMARKS

- This function is supported on Windows only.

A.3.4 WDU_GetDeviceInfo()

PURPOSE

• Gets configuration information from a device, including all the device descriptors.

NOTE: The caller to this function is responsible for calling `WDU_PutDeviceInfo()` [A.3.5] in order to free the `*ppDeviceInfo` pointer returned by the function.

PROTOTYPE

```
DWORD WDU_GetDeviceInfo(
    WDU_DEVICE_HANDLE hDevice,
    WDU_DEVICE **ppDeviceInfo);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ ppDeviceInfo	WDU_DEVICE**	Output

DESCRIPTION

Name	Description
hDevice	A unique identifier for a device/interface
ppDeviceInfo	Pointer to pointer to a USB device information structure [A.4.3]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.5 WDU_PutDeviceInfo()

PURPOSE

• Receives a device information pointer, allocated with a previous `WDU_GetDeviceInfo()` [A.3.4] call, in order to perform the necessary cleanup.

PROTOTYPE

```
void WDU_PutDeviceInfo(WDU_DEVICE *pDeviceInfo);
```

PARAMETERS

Name	Type	Input/Output
➢ pDeviceInfo	WDU_DEVICE*	Input

DESCRIPTION

Name	Description
pDeviceInfo	Pointer to a USB device information structure [A.4.3], as returned by a previous call to `WDU_GetDeviceInfo()` [A.3.4]

RETURN VALUE

None

A.3.6 WDU_Uninit()

PURPOSE

• Stops listening to devices matching a given criteria and un-registers the notification callbacks for these devices.

PROTOTYPE

```
void WDU_Uninit(WDU_DRIVER_HANDLE hDriver);
```

PARAMETERS

Name	Type	Input/Output
➢ hDriver	WDU_DRIVER_HANDLE	Input

DESCRIPTION

Name	Description
hDriver	Handle to the registration received from `WDU_Init()` [A.3.1]

RETURN VALUE

None

A.3.7 WDU_Transfer()

PURPOSE

• Transfers data to or from a device.

PROTOTYPE

```
DWORD WDU_Transfer(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    PBYTE pSetupPacket,
    DWORD dwTimeout);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ dwPipeNum	DWORD	Input
➢ fRead	DWORD	Input
➢ dwOptions	DWORD	Input
➢ pBuffer	PVOID	Input
➢ dwBufferSize	DWORD	Input
➢ pdwBytesTransferred	PDWORD	Output
➢ pSetupPacket	PBYTE	Input
➢ dwTimeout	DWORD	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface received from `WDU_Init()` [A.3.1]
dwPipeNum	The number of the pipe through which the data is transferred
fRead	`TRUE` for read, `FALSE` for write
dwOptions	A bit-mask, which can be comprises of any of the following flags: • **`USB_ISOCH_NOASAP`** – For isochronous data transfers. Setting this option instructs the lower USB stack driver (**usbd.sys**) to use a preset frame number (instead of the next available frame) while performing the data transfer. Use this flag if you notice unused frames during the transfer, on low-speed or full-speed devices (USB 1.1 only) and on Windows only (excluding WinCE). • **`USB_ISOCH_RESET`** – resets the isochronous pipe before the data transfer. It also resets the pipe after minor errors (consequently allowing to continue with the transfer). • **`USB_ISOCH_FULL_PACKETS_ONLY`** – when set, do not transfer less than packet size on isochronous pipes.
pBuffer	Address of the data buffer
dwBufferSize	Number of bytes to transfer. The buffer size is not limited to the device's maximum packet size; therefore, you can use larger buffers by setting the buffer size to a multiple of the maximum packet size. Use large buffers to reduce the number of context switches and thereby improve performance.
pdwBytesTransferred	Number of bytes actually transferred
pSetupPacket	An 8-byte packet to transfer to control pipes
dwTimeout	Timeout interval of the transfer, in milliseconds (*ms*). If **`dwTimeout`** is zero, the function's timeout interval never elapses (infinite wait).

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

REMARKS

- The resolution of the timeout (the dwTimeout parameter) is according to the operating system scheduler's time slot. For example, in Windows the timeout's resolution is 10 milliseconds (*ms*).

A.3.8 WDU_Wakeup()

PURPOSE

• Enables/Disables the wakeup feature.

PROTOTYPE

```
DWORD WDU_Wakeup(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwOptions);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ dwOptions	DWORD	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface
dwOptions	Can be either: • **WDU_WAKEUP_ENABLE** – enable wakeup OR: • **WDU_WAKEUP_DISABLE** – disable wakeup

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.9 WDU_TransferDefaultPipe()

PURPOSE

• Transfers data to or from a device through the default pipe.

PROTOTYPE

```
DWORD WDU_TransferDefaultPipe(
    WDU_DEVICE_HANDLE hDevice,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    PBYTE pSetupPacket,
    DWORD dwTimeout);
```

PARAMETERS

See parameters of `WDU_Transfer()` [A.3.7].
Note that `dwPipeNum` is not a parameter of this function.

DESCRIPTION

See description of `WDU_Transfer()` [A.3.7] .

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.10 WDU_TransferBulk()

PURPOSE

• Performs bulk data transfer to or from a device.

PROTOTYPE

```
DWORD WDU_TransferBulk(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    DWORD dwTimeout);
```

PARAMETERS

See parameters of `WDU_Transfer()` [A.3.7].
Note that `pSetupPacket` is not a parameter of this function.

DESCRIPTION

See description of `WDU_Transfer()` [A.3.7] .

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.11 WDU_TransferIsoch()

PURPOSE

• Performs isochronous data transfer to or from a device.

PROTOTYPE

```
DWORD WDU_TransferIsoch(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    DWORD dwTimeout);
```

PARAMETERS

See parameters of `WDU_Transfer()` [A.3.7].
Note that `pSetupPacket` is not a parameter of this function.

DESCRIPTION

See description of `WDU_Transfer()` [A.3.7].

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.12 WDU_TransferInterrupt()

PURPOSE

• Performs interrupt data transfer to or from a device.

PROTOTYPE

```
DWORD WDU_TransferInterrupt(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum,
    DWORD fRead,
    DWORD dwOptions,
    PVOID pBuffer,
    DWORD dwBufferSize,
    PDWORD pdwBytesTransferred,
    DWORD dwTimeout);
```

PARAMETERS

See parameters of `WDU_Transfer()` [A.3.7].
Note that `pSetupPacket` is not a parameter of this function.

DESCRIPTION

See description of `WDU_Transfer()` [A.3.7] .

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.13 WDU_HaltTransfer()

PURPOSE

• Halts the transfer on the specified pipe (only one simultaneous transfer per pipe is allowed by WinDriver).

PROTOTYPE

```
DWORD WDU_HaltTransfer(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ dwPipeNum	DWORD	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface
dwPipeNum	The number of the pipe

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.3.14 WDU_ResetPipe()

PURPOSE

• Resets a pipe by clearing both the halt condition on the host side of the pipe and the stall condition on the endpoint. This function is applicable for all pipes except pipe00.

PROTOTYPE

```
DWORD WDU_ResetPipe(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwPipeNum);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ dwPipeNum	DWORD	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface
dwPipeNum	The pipe's number

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

REMARKS

- This function should be used if a pipe is halted, in order to clear the halt.

A.3.15 WDU_ResetDevice()

PURPOSE

• Resets a device to help recover from an error, when a device is marked as connected but is not enabled.

PROTOTYPE

```
DWORD WDU_ResetDevice(
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwOptions);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ dwOptions	DWORD	Input

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface.
dwOptions	Can be either zero or: • **`WD_USB_HARD_RESET`** – reset the device even if it is not disabled. After using this option it is advised to set the interface of the device, using `WDU_SetInterface()` [A.3.2].

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

REMARKS

- `WDU_ResetDevice()` is supported only on Windows and Windows CE 5.0.
- This function issues a request from the Windows USB driver to reset a hub port, provided the Windows USB driver supports this feature.

A.3.16 WDU_GetLangIDs()

PURPOSE

• Reads a list of supported language IDs and/or the number of supported language IDs from a device.

PROTOTYPE

```
DWORD DLLCALLCONV WDU_GetLangIDs(
    WDU_DEVICE_HANDLE hDevice,
    PBYTE pbNumSupportedLangIDs,
    WDU_LANGID *pLangIDs,
    BYTE bNumLangIDs);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ pbNumSupportedLangIDs	PBYTE	Output
➢ pLangIDs	WDU_LANGID*	Output
➢ bNumLangIDs	BYTE	Input

DESCRIPTION

Name	**Description**
hDevice	A unique identifier for the device/interface
pbNumSupportedLangIDs	Parameter to receive number of supported language IDs
pLangIDs	Array of language IDs. If **`bNumLangIDs`** is not zero the function will fill this array with the supported language IDs for the device.
bNumLangIDs	Number of IDs in the pLangIDs array

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

Remarks

- If **`dwNumLangIDs`** is zero the function will return only the number of supported language IDs (in **`pbNumSupportedLangIDs`**) but will not update the language IDs array (**`pLangIDs`**) with the actual IDs. For this usage **`pLangIDs`** can be `NULL` (since it is not referenced) but **`pbNumSupportedLangIDs`** must not be `NULL`.
- **`pbNumSupportedLangIDs`** can be `NULL` if the user only wants to receive the list of supported language IDs and not the number of supported IDs. In this case **`bNumLangIDs`** cannot be zero and **`pLangIDs`** cannot be `NULL`.
- If the device does not support any language IDs the function will return success. The caller should therefore check the value of **`*pbNumSupportedLangIDs`** after the function returns.
- If the size of the **`pLangIDs`** array (**`bNumLangIDs`**) is smaller than the number of IDs supported by the device (**`*pbNumSupportedLangIDs`**), the function will read and return only the first **`bNumLangIDs`** supported language IDs.

A.3.17 WDU_GetStringDesc()

PURPOSE

• Reads a string descriptor from a device by string index.

PROTOTYPE

```
DWORD DLLCALLCONV WDU_GetStringDesc(
    WDU_DEVICE_HANDLE hDevice,
    BYTE bStrIndex,
    PCHAR pcDescStr,
    DWORD dwSize,
    WDU_LANGID langID);
```

PARAMETERS

Name	Type	Input/Output
➢ hDevice	WDU_DEVICE_HANDLE	Input
➢ bStrIndex	BYTE	Input
➢ pbBuf	PBYTE	Output
➢ dwBufSize	DWORD	Input
➢ langID	WDU_LANGID	Input
➢ pdwDescSize	PDWORD	Output

DESCRIPTION

Name	Description
hDevice	A unique identifier for the device/interface
bStrIndex	A string index
pbBuf	The read string descriptor (the descriptor is returned as a bytes array)
dwBufSize	The size of `pbBuf`
langID	The language ID to be used in the get string descriptor request that is sent to the device. If the `langID` param is 0, the function will use the first supported language ID returned from the device (if exists).
pdwDescSize	If not `NULL`, will be updated with the size of the returned descriptor

Return Value

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

Remarks

- If **`pbBuf`** is not large enough to hold the string descriptor (**`dwBufSize`** < **`*pdwDescSize`**), the returned descriptor will be truncated to **`dwBufSize`** bytes.

A.4 USB Structures

The following figure depicts the structure hierarchy used by WinDriver's USB API. The arrays situated in each level of the hierarchy may contain more elements than are depicted in the diagram. Arrows are used to represent pointers. In the interest of clarity, only one structure at each level of the hierarchy is depicted in full detail (with all of its elements listed and pointers from it pictured).

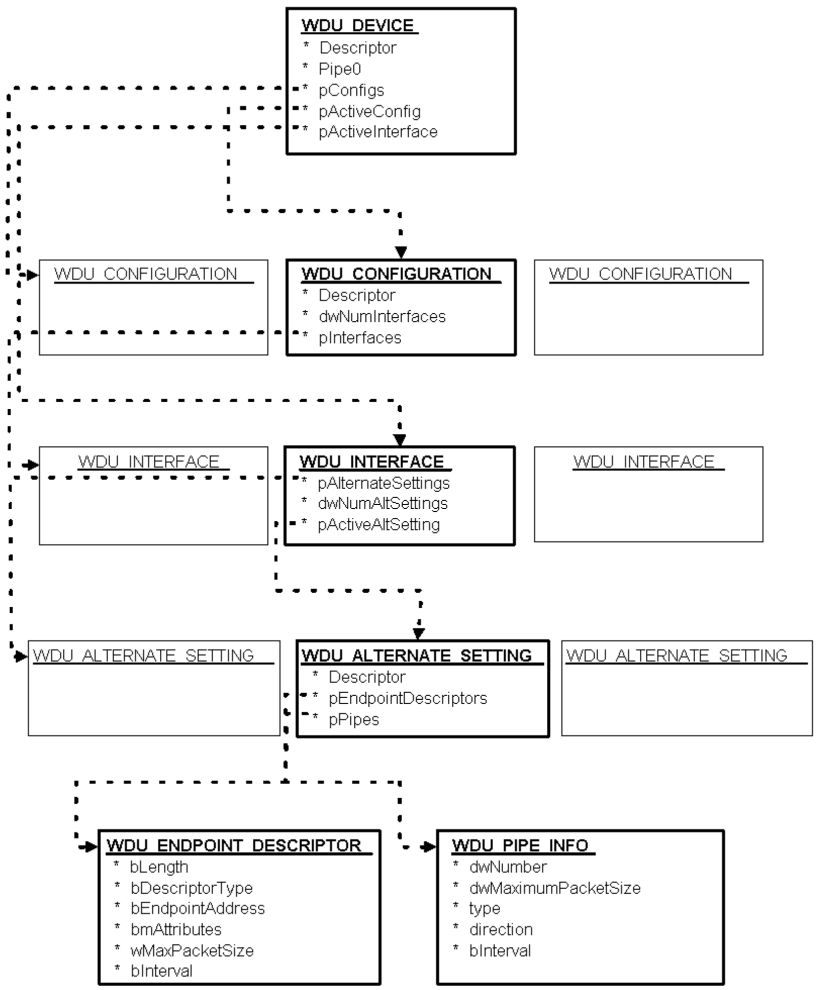


Figure A.2: WinDriver USB Structures

A.4.1 WDU_MATCH_TABLE

USB match table structure:

> **NOTE**
> (*) For all field members, if value is set to zero – match all.

Name	Type	Description
wVendorId	WORD	Required USB Vendor ID to detect, as assigned by USB-IF (*)
wProductId	WORD	Required USB Product ID to detect, as assigned by the product manufacturer (*)
bDeviceClass	BYTE	The device's class code, as assigned by USB-IF (*)
bDeviceSubClass	BYTE	The device's sub-class code, as assigned by USB-IF (*)
bInterfaceClass	BYTE	The interface's class code, as assigned by USB-IF (*)
bInterfaceSubClass	BYTE	The interface's sub-class code, as assigned by USB-IF (*)
bInterfaceProtocol	BYTE	The interface's protocol code, as assigned by USB-IF (*)

A.4.2 WDU_EVENT_TABLE

USB events table structure:

Name	Type	Description
pfDeviceAttach	WDU_ATTACH_CALLBACK	Will be called by WinDriver when a device is attached
pfDeviceDetach	WDU_DETACH_CALLBACK	Will be called by WinDriver when a device is detached
pfPowerChange	WDU_POWER_CHANGE_CALLBACK	Will be called by WinDriver when there is a change in a device's power state
pUserData	PVOID	Pointer to user-mode data to be passed to the callbacks

A.4.3 WDU_DEVICE

USB device information structure:

Name	Type	Description
Descriptor	WDU_DEVICE_DESCRIPTOR	Device descriptor information structure [A.4.7]
Pipe0	WDU_PIPE_INFO	Pipe information structure [A.4.11] for the device's default pipe (Pipe 0)
pConfigs	WDU_CONFIGURATION*	Pointer to the device's configuration information structure [A.4.4]
pActiveConfig	WDU_CONFIGURATION*	Pointer to a configuration information structure [A.4.4] for the device's active configuration
pActiveInterface	WDU_INTERFACE* [WD_USB_MAX_INTERFACES]	Array of pointers to interface information structures [A.4.5] for the device's active interfaces

A.4.4 WDU_CONFIGURATION

Configuration information structure:

Name	Type	Description
Descriptor	WDU_CONFIGURATION_DESCRIPTOR	Configuration descriptor information structure [A.4.8]
dwNumInterfaces	DWORD	Number of interfaces supported by this configuration
pInterfaces	WDU_INTERFACE*	Pointer to the beginning of an array of interface information structures [A.4.5] for the configuration's interfaces

A.4.5 WDU_INTERFACE

Interface information structure:

Name	Type	Description
pAlternateSettings	WDU_ALTERNATE_SETTING*	Pointer to the beginning of an array of alternate setting information structures [A.4.6] for the interface's alternate settings
dwNumAltSettings	DWORD	Number of alternate settings supported by this interface
pActiveAltSetting	WDU_ALTERNATE_SETTING*	Pointer to an alternate setting information structure [A.4.6] for the interface's active alternate setting

A.4.6 WDU_ALTERNATE_SETTING

Alternate setting information structure:

Name	Type	Description
Descriptor	WDU_INTERFACE_DESCRIPTOR	Interface descriptor information structure [A.4.9]
pEndpointDescriptors	WDU_ENDPOINT_DESCRIPTOR*	Pointer to the beginning of an array of endpoint descriptor information structures [A.4.10] for the alternate setting's endpoints
pPipes	WDU_PIPE_INFO*	Pointer to the beginning of an array of pipe information structures [A.4.11] for the alternate setting's pipes

A.4.7 WDU_DEVICE_DESCRIPTOR

USB device descriptor information structure:

Name	**Type**	**Description**
bLength	UCHAR	Size, in bytes, of the descriptor (18 bytes)
bDescriptorType	UCHAR	Device descriptor (0x01)
bcdUSB	USHORT	Number of the USB specification with which the device complies
bDeviceClass	UCHAR	The device’s class
bDeviceSubClass	UCHAR	The device’s sub-class
bDeviceProtocol	UCHAR	The device’s protocol
bMaxPacketSize0	UCHAR	Maximum size of transferred packets
idVendor	USHORT	Vendor ID, as assigned by USB-IF
idProduct	USHORT	Product ID, as assigned by the product manufacturer
bcdDevice	USHORT	Device release number
iManufacturer	UCHAR	Index of manufacturer string descriptor
iProduct	UCHAR	Index of product string descriptor
iSerialNumber	UCHAR	Index of serial number string descriptor
bNumConfigurations	UCHAR	Number of possible configurations

A.4.8 WDU_CONFIGURATION_DESCRIPTOR

USB configuration descriptor information structure:

Name	Type	Description
bLength	UCHAR	Size, in bytes, of the descriptor
bDescriptorType	UCHAR	Configuration descriptor (0x02)
wTotalLength	USHORT	Total length, in bytes, of data returned
bNumInterfaces	UCHAR	Number of interfaces
bConfigurationValue	UCHAR	Configuration number
iConfiguration	UCHAR	Index of string descriptor that describes this configuration
bmAttributes	UCHAR	Power settings for this configuration: • $D6$ – self-powered • $D5$ – remote wakeup (allows device to wake up the host)
MaxPower	UCHAR	Maximum power consumption for this configuration, in $2mA$ units

A.4.9 WDU_INTERFACE_DESCRIPTOR

USB interface descriptor information structure:

Name	Type	Description
bLength	UCHAR	Size, in bytes, of the descriptor (9 bytes)
bDescriptorType	UCHAR	Interface descriptor (0x04)
bInterfaceNumber	UCHAR	Interface number
bAlternateSetting	UCHAR	Alternate setting number
bNumEndpoints	UCHAR	Number of endpoints used by this interface
bInterfaceClass	UCHAR	The interface's class code, as assigned by USB-IF
bInterfaceSubClass	UCHAR	The interface's sub-class code, as assigned by USB-IF
bInterfaceProtocol	UCHAR	The interface's protocol code, as assigned by USB-IF
iInterface	UCHAR	Index of string descriptor that describes this interface

A.4.10 WDU_ENDPOINT_DESCRIPTOR

USB endpoint descriptor information structure:

Name	Type	Description
bLength	UCHAR	Size, in bytes, of the descriptor (7 bytes)
bDescriptorType	UCHAR	Endpoint descriptor (0x05)
bEndpointAddress	UCHAR	Endpoint address: Use bits 0-3 for endpoint number, set bits 4-6 to zero (0), and set bit 7 to zero (0) for outbound data and one (1) for inbound data (ignored for control endpoints)
bmAttributes	UCHAR	Specifies the transfer type for this endpoint (control, interrupt, isochronous or bulk). See the USB specification for further information.
wMaxPacketSize	USHORT	Maximum size of packets this endpoint can send or receive
bInterval	UCHAR	Interval, in frame counts, for polling endpoint data transfers. Ignored for bulk and control endpoints. Must equal 1 for isochronous endpoints. May range from 1 to 255 for interrupt endpoints.

A.4.11 WDU_PIPE_INFO

USB pipe information structure:

Name	Type	Description
dwNumber	DWORD	Pipe number; Zero for default pipe
dwMaximumPacketSize	DWORD	Maximum size of packets that can be transferred using this pipe
type	DWORD	Transfer type for this pipe
direction	DWORD	Direction of the transfer: • **`USB_DIR_IN`** or **`USB_DIR_OUT`** for isochronous, bulk or interrupt pipes. • **`USB_DIR_IN_OUT`** for control pipes.
dwInterval	DWORD	Interval in milliseconds (*ms*). Relevant only to interrupt pipes.

A.5 General WD_xxx Functions

A.5.1 Calling Sequence WinDriver – General Use

The following is a typical calling sequence for the WinDriver API.

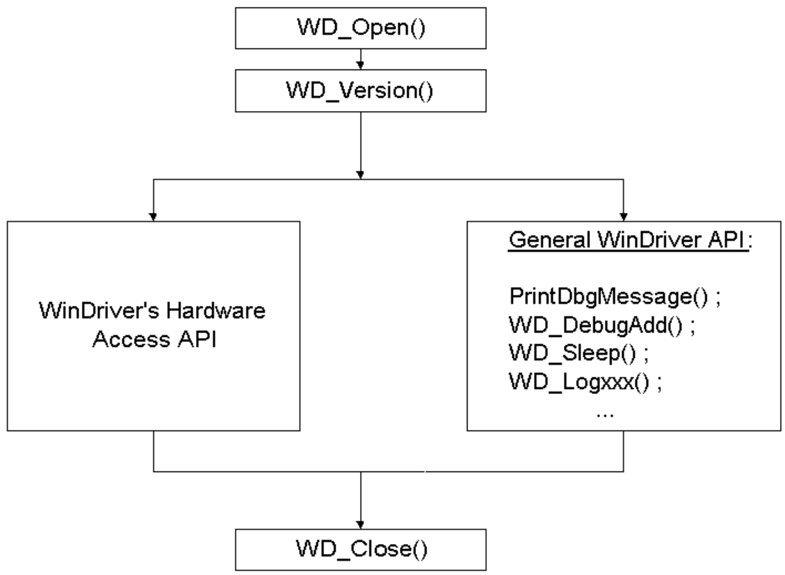


Figure A.3: WinDriver API Calling Sequence

NOTES

- We recommend calling the WinDriver function `WD_Version()` [A.5.3] after calling `WD_Open()` [A.5.2] and before calling any other WinDriver function. Its purpose is to return the WinDriver kernel module (windrvr) version number, thus providing the means to verify that your application is version compatible with the WinDriver kernel module.
- `WD_DebugAdd()` [A.5.6] and `WD_Sleep()` [A.5.8] can be called anywhere after `WD_Open()`.

A.5.2 WD_Open()

PURPOSE

• Opens a handle to access the WinDriver kernel module. The handle is used by all WinDriver APIs, and therefore must be called before any other WinDriver API is called.

PROTOTYPE

```
HANDLE WD_Open(void);
```

RETURN VALUE

The handle to the WinDriver kernel module.
If device could not be opened, returns `INVALID_HANDLE_VALUE`.

REMARKS

- If you are a registered user, please refer to the documentation of `WD_License()` [A.5.9] for an example of how to register your WinDriver license.

EXAMPLE

```
HANDLE hWD;

hWD = WD_Open();
if (hWD == INVALID_HANDLE_VALUE)
{
    printf("Cannot open WinDriver device\n");
}
```

A.5.3 WD_Version()

PURPOSE

• Returns the version number of the WinDriver kernel module currently running.

PROTOTYPE

```
DWORD WD_Version(
    HANDLE hWD,
    WD_VERSION *pVer);
```

PARAMETERS

Name	Type	Input/Output
➢ hWD	HANDLE	Input
➢ pVer	WD_VERSION*	
❑ dwVer	DWORD	Output
❑ cVer	CHAR[100]	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from `WD_Open()` [A.5.2]
pVer	Pointer to a WinDriver version information structure:
dwVer	The version number
cVer	Version information string

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

EXAMPLE

```
WD_VERSION ver;

BZERO(ver);
WD_Version(hWD, &ver);
printf("%s\n", ver.cVer);
if (ver.dwVer < WD_VER)
{
    printf("Error - incorrect WinDriver version\n");
}
```

A.5.4 WD_Close()

PURPOSE

• Closes the access to the WinDriver kernel module.

PROTOTYPE

```
void WD_Close(HANDLE hWD);
```

PARAMETERS

Name	Type	Input/Output
➢ hWD	HANDLE	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from `WD_Open()` [A.5.2]

RETURN VALUE

None

REMARKS

- This function must be called when you finish using WinDriver kernel module.

EXAMPLE

```
WD_Close(hWD);
```

A.5.5 WD_Debug()

PURPOSE

• Sets debugging level for collecting debug messages.

PROTOTYPE

```
DWORD WD_Debug(
    HANDLE hWD,
    WD_DEBUG *pDebug);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ hWD	HANDLE	Input
➢ pDebug	WD_DEBUG*	Input
❏ dwCmd	DWORD	Input
❏ dwLevel	DWORD	Input
❏ dwSection	DWORD	Input
❏ dwLevelMessageBox	DWORD	Input
❏ dwBufferSize	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from `WD_Open()` [A.5.2]
pDebug	Pointer to a debug information structure:
dwCmd	Debug command: Set filter, Clear buffer, etc. For more details please refer to `DEBUG_COMMAND` in **windrvr.h**.
dwLevel	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the debugging level to collect: Error, Warning, Info, Trace. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
dwSection	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the sections to collect: I/O, Memory, Interrupt, etc. Use `S_ALL` for all. For more details please refer to `DEBUG_SECTION` in **windrvr.h**.
dwLevelMessageBox	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the debugging level to print in a message box. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
dwBufferSize	Used for `dwCmd=DEBUG_SET_BUFFER`. The size of buffer in the kernel.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

EXAMPLE

```
WD_DEBUG dbg;

BZERO(dbg);
dbg.dwCmd = DEBUG_SET_FILTER;
dbg.dwLevel = D_ERROR;
dbg.dwSection = S_ALL;
dbg.dwLevelMessageBox = D_ERROR;

WD_Debug(hWD, &dbg);
```

A.5.6 WD_DebugAdd()

PURPOSE

• Sends debug messages to the debug log. Used by the driver code.

PROTOTYPE

```
DWORD WD_DebugAdd(
    HANDLE hWD,
    WD_DEBUG_ADD *pData);
```

PARAMETERS

Name	Type	Input/Output
➢ hWD	HANDLE	Input
➢ pData	WD_DEBUG_ADD*	
❏ dwLevel	DWORD	Input
❏ dwSection	DWORD	Input
❏ pcBuffer	CHAR [256]	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from `WD_Open()` [A.5.2]
pData	Pointer to an additional debug information structure:
dwLevel	Assigns the level in the Debug Monitor, in which the data will be declared. If `dwLevel` is zero, `D_ERROR` will be declared. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
dwSection	Assigns the section in the Debug Monitor, in which the data will be declared. If `dwSection` is zero, `S_MISC` section will be declared. For more details please refer to `DEBUG_SECTION` in **windrvr.h**.
pcBuffer	The string to copy into the message log.

Return Value

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

Example

```
WD_DEBUG_ADD add;

BZERO(add);
add.dwLevel = D_WARN;
add.dwSection = S_MISC;
sprintf(add.pcBuffer, "This message will be displayed in "
    "the debug monitor\n");
WD_DebugAdd(hWD, &add);
```

A.5.7 WD_DebugDump()

PURPOSE

• Retrieves debug messages buffer.

PROTOTYPE

```
DWORD WD_DebugDump(
    HANDLE hWD,
    WD_DEBUG_DUMP *pDebugDump);
```

PARAMETERS

Name	Type	Input/Output
➢ hWD	HANDLE	Input
➢ pDebug	WD_DEBUG_DUMP*	Input
❏ pcBuffer	PCHAR	Input/Output
❏ dwSize	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from `WD_Open()` [A.5.2]
pDebugDump	Pointer to a debug dump information structure:
pcBuffer	Buffer to receive debug messages
dwSize	Size of buffer in bytes

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

EXAMPLE

```
char buffer[1024];
WD_DEBUG_DUMP dump;
dump.pcBuffer=buffer;
dump.dwSize = sizeof(buffer);
WD_DebugDump(hWD, &dump);
```

A.5.8 WD_Sleep()

PURPOSE

• Delays execution for a specific duration of time.

PROTOTYPE

```
DWORD WD_Sleep(
    HANDLE hWD,
    WD_SLEEP *pSleep);
```

PARAMETERS

Name	Type	Input/Output
➢ hWD	HANDLE	Input
➢ pSleep	WD_SLEEP*	
❏ dwMicroSeconds	DWORD	Input
❏ dwOptions	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from `WD_Open()` [A.5.2]
pSleep	Pointer to a sleep information structure:
dwMicroSeconds	Sleep time in microseconds – 1/1,000,000 of a second.
dwOptions	A bit-mask, which can be set to either of the following: • **`Zero (0)`** – Busy sleep (default) OR: • **`SLEEP_NON_BUSY`** – Delay execution without consuming CPU resources. (Not relevant for under 17,000 micro seconds. Less accurate than busy sleep).

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

REMARKS

- Example usage: to access slow response hardware.

EXAMPLE

```
WD_Sleep slp;

BZERO(slp);
slp.dwMicroSeconds = 200;
WD_Sleep(hWD, &slp);
```

A.5.9 WD_License()

PURPOSE

• Transfers the license string to the WinDriver kernel module and returns information regarding the license type of the specified license string.

NOTE: When using the `WDU` USB APIs [A.1] your WinDriver license registration is done via the call to `WDU_Init()` [A.3.1], so you do not need to call `WD_License()` directly from your code.

PROTOTYPE

```
DWORD WD_License(
    HANDLE hWD,
    WD_LICENSE *pLicense);
```

PARAMETERS

Name	Type	Input/Output
➢ hWD	HANDLE	Input
➢ pLicense	WD_LICENSE*	
❏ cLicense	CHAR[]	Input
❏ dwLicense	DWORD	Output
❏ dwLicense2	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from `WD_Open()` [A.5.2]
pLicense	Pointer to a WinDriver license information structure:
cLicense	A buffer to contain the license string that is to be transferred to the WinDriver kernel module. If an empty string is transferred, then WinDriver kernel module returns the current license type to the parameter `dwLicense`.
dwLicense	Returns the license type of the specified license string (`cLicnese`). The return value is a bit-mask of license flags, defined as an enum in **windrvr.h**. Zero signifies an invalid license string. Additional flags for determining the license type are returned in **`dwLicense2`**, if needed.
dwLicense2	Returns additional flags for determining the license type, if `dwLicense` cannot hold all the relevant information (otherwise – zero)

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

REMARKS

- When using a registered version, this function must be called before any other WinDriver API call, apart from `WD_Open()` [A.5.2], in order to register the license from the code.

EXAMPLE

Example usage: Add registration routine to your application:

```
DWORD RegisterWinDriver()
{
    HANDLE hWD;
    WD_LICENSE lic;
    DWORD dwStatus = WD_INVALID_HANDLE;

    hWD = WD_Open();
    if (hWD!=INVALID_HANDLE_VALUE)
    {
        BZERO(lic);
        /* Replace the following string with your license string: */
        strcpy(lic.cLicense, "12345abcde12345.CompanyName");
        dwStatus = WD_License(hWD, &lic);
        WD_Close(hWD);
    }

    return dwStatus;
}
```

A.6 User-Mode Utility Functions

This section describes a number of user-mode utility functions you will find useful for implementing various tasks. These utility functions are multi-platform, implemented on all operating systems supported by WinDriver.

A.6.1 Stat2Str()

PURPOSE

• Retrieves the status string that corresponds to a status code.

PROTOTYPE

```
const char *Stat2Str(DWORD dwStatus);
```

PARAMETERS

Name	Type	Input/Output
➢ dwStatus	DWORD	Input

DESCRIPTION

Name	Description
dwStatus	A numeric status code

RETURN VALUE

Returns the verbal status description (string) that corresponds to the specified numeric status code.

REMARKS

See section A.7 for a complete list of status codes and strings.

A.6.2 get_os_type()

PURPOSE

- Retrieves the type of the operating system.

PROTOTYPE

```
OS_TYPE get_os_type(void);
```

RETURN VALUE

Returns the type of the operating system.
If the operating system type is not detected, returns `OS_CAN_NOT_DETECT`.

A.6.3 ThreadStart()

PURPOSE

• Creates a thread.

PROTOTYPE

```
DWORD ThreadStart(
    HANDLE *phThread,
    HANDLER_FUNC pFunc,
    void *pData);
```

PARAMETERS

Name	Type	Input/Output
➢ phThread	HANDLE*	Output
➢ pFunc	HANDLER_FUNC	Input
➢ pData	VOID*	Input

DESCRIPTION

Name	Description
phThread	Returns the handle to the created thread
pFunc	Starting address of the code that the new thread is to execute
pData	Pointer to the data to be passed to the new thread

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.6.4 ThreadWait()

PURPOSE

• Waits for a thread to exit.

PROTOTYPE

```
void ThreadWait(HANDLE hThread);
```

PARAMETERS

Name	Type	Input/Output
➢ hThread	HANDLE	Input

DESCRIPTION

Name	Description
hThread	The handle to the thread whose completion is awaited

RETURN VALUE

None

A.6.5 OsEventCreate()

PURPOSE

• Creates an event object.

PROTOTYPE

```
DWORD OsEventCreate(HANDLE *phOsEvent);
```

PARAMETERS

Name	Type	Input/Output
➢ phOsEvent	HANDLE*	Output

DESCRIPTION

Name	Description
phOsEvent	The pointer to a variable that receives a handle to the newly created event object

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.6.6 OsEventClose()

PURPOSE

• Closes a handle to an event object.

PROTOTYPE

```
void OsEventClose(HANDLE hOsEvent);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ hOsEvent	HANDLE	Input

DESCRIPTION

Name	**Description**
hOsEvent	The handle to the event object to be closed

RETURN VALUE

None

A.6.7 OsEventWait()

PURPOSE

• Waits until a specified event object is in the signaled state or the time-out interval elapses.

PROTOTYPE

```
DWORD OsEventWait(
    HANDLE hOsEvent,
    DWORD dwSecTimeout);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ hOsEvent	HANDLE	Input
➢ dwSecTimeout	DWORD	Input

DESCRIPTION

Name	**Description**
hOsEvent	The handle to the event object
dwSecTimeout	Time-out interval of the event, in seconds. A time-out value of zero signifies an infinite wait.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.6.8 OsEventSignal()

PURPOSE

• Sets the specified event object to the signaled state.

PROTOTYPE

```
DWORD OsEventSignal(HANDLE hOsEvent);
```

PARAMETERS

Name	Type	Input/Output
➢ hOsEvent	HANDLE	Input

DESCRIPTION

Name	Description
hOsEvent	The handle to the event object

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.6.9 OsEventReset()

PURPOSE

• Resets the specified event object to the non-signaled state.

PROTOTYPE

```
DWORD OsEventReset(HANDLE hOsEvent);
```

PARAMETERS

Name	Type	Input/Output
➢ hOsEvent	HANDLE	Input

DESCRIPTION

Name	Description
hOsEvent	The handle to the event object

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.6.10 OsMutexCreate()

PURPOSE

- Creates a mutex object.

PROTOTYPE

```
DWORD OsMutexCreate(HANDLE *phOsMutex);
```

PARAMETERS

Name	Type	Input/Output
➢ phOsMutex	HANDLE*	Output

DESCRIPTION

Name	Description
phOsMutex	The pointer to a variable that receives a handle to the newly created mutex object

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.6.11 OsMutexClose()

PURPOSE

- Closes a handle to a mutex object.

PROTOTYPE

```
void OsMutexClose(HANDLE hOsMutex);
```

PARAMETERS

Name	Type	Input/Output
➢hOsMutex	HANDLE	Input

DESCRIPTION

Name	Description
hOsMutex	The handle to the mutex object to be closed

RETURN VALUE

None

A.6.12 OsMutexLock()

PURPOSE

• Locks the specified mutex object.

PROTOTYPE

```
DWORD OsMutexLock(HANDLE hOsMutex);
```

PARAMETERS

Name	Type	Input/Output
➢ hOsMutex	HANDLE	Input

DESCRIPTION

Name	Description
hOsMutex	The handle to the mutex object to be locked

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.6.13 OsMutexUnlock()

PURPOSE

• Releases (unlocks) a locked mutex object.

PROTOTYPE

```
DWORD OsMutexUnlock(HANDLE hOsMutex);
```

PARAMETERS

Name	Type	Input/Output
➢hOsMutex	HANDLE	Input

DESCRIPTION

Name	Description
hOsMutex	The handle to the mutex object to be unlocked

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.6.14 PrintDbgMessage()

PURPOSE

• Sends debug messages to the debug monitor.

PROTOTYPE

```
void PrintDbgMessage(
    DWORD dwLevel,
    DWORD dwSection,
    const char *format
    [, argument]...);
```

PARAMETERS

Name	Type	Input/Output
➢ dwLevel	DWORD	Input
➢ dwSection	DWORD	Input
➢ format	const char*	Input
➢ argument		Input

DESCRIPTION

Name	Description
dwLevel	Assigns the level in the Debug Monitor, in which the data will be declared. If zero, `D_ERROR` will be declared. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
dwSection	Assigns the section in the Debug Monitor, in which the data will be declared. If zero, `S_MISC` will be declared. For more details please refer to DEBUG_SECTION in **windrvr.h**.
format	Format-control string
argument	Optional arguments, limited to 256 bytes

RETURN VALUE

None

A.6.15 WD_LogStart()

PURPOSE

• Opens a log file.

PROTOTYPE

```
DWORD WD_LogStart(
    const char *sFileName,
    const char *sMode);
```

PARAMETERS

Name	Type	Input/Output
➢ sFileName	const char*	Input
➢ sMode	const char*	Input

DESCRIPTION

Name	Description
sFileName	Name of log file to be opened
sMode	Type of access permitted. For example, NULL or **w** opens an empty file for writing, and if the given file exists, its contents are destroyed; **a** opens a file for writing at the end of the file (i.e. append).

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

REMARKS

- Once a log file is opened, all API calls are logged in this file. You may add your own printouts to the log file by calling `WD_LogAdd()` [A.6.17].

A.6.16 WD_LogStop()

PURPOSE

• Closes a log file.

PROTOTYPE

```
VOID WD_LogStop( void );
```

RETURN VALUE

None

A.6.17 WD_LogAdd()

PURPOSE

• Adds user printouts into log file.

PROTOTYPE

```
VOID DLLCALLCONV WD_LogAdd(
    const char *sFormat
    [, argument ]...);
```

PARAMETERS

Name	Type	Input/Output
➢ sFormat	const char*	Input
➢ argument		Input

DESCRIPTION

Name	Description
sFormat	Format-control string
argument	Optional format arguments

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.7].

A.7 WinDriver Status Codes

A.7.1 Introduction

Most of the WinDriver functions return a status code, where zero (`WD_STATUS_SUCCESS`) means success and a non-zero value means failure.
The `Stat2Str()` functions can be used to retrieve the status description string for a given status code. The status codes and their descriptive strings are listed below.

A.7.2 Status Codes Returned by WinDriver

Status Code	Description
WD_STATUS_SUCCESS	Success
WD_STATUS_INVALID_WD_HANDLE	Invalid WinDriver handle
WD_WINDRIVER_STATUS_ERROR	Error
WD_INVALID_HANDLE	Invalid handle
WD_INVALID_PIPE_NUMBER	Invalid pipe number
WD_READ_WRITE_CONFLICT	Conflict between read and write operations
WD_ZERO_PACKET_SIZE	Packet size is zero
WD_INSUFFICIENT_RESOURCES	Insufficient resources
WD_UNKNOWN_PIPE_TYPE	Unknown pipe type
WD_SYSTEM_INTERNAL_ERROR	Internal system error
WD_DATA_MISMATCH	Data mismatch
WD_NO_LICENSE	No valid license
WD_NOT_IMPLEMENTED	Function not implemented
WD_FAILED_ENABLING_INTERRUPT	Failed enabling interrupt
WD_INTERRUPT_NOT_ENABLED	Interrupt not enabled
WD_RESOURCE_OVERLAP	Resource overlap
WD_DEVICE_NOT_FOUND	Device not found
WD_WRONG_UNIQUE_ID	Wrong unique ID
WD_OPERATION_ALREADY_DONE	Operation already done
WD_USB_DESCRIPTOR_ERROR	USB descriptor error
WD_SET_CONFIGURATION_FAILED	Set configuration operation failed
WD_CANT_OBTAIN_PDO	Cannot obtain PDO
WD_TIME_OUT_EXPIRED	Timeout expired
WD_IRP_CANCELED	IRP operation cancelled
WD_FAILED_USER_MAPPING	Failed to map in user space
WD_FAILED_KERNEL_MAPPING	Failed to map in kernel space
WD_NO_RESOURCES_ON_DEVICE	No resources on the device
WD_NO_EVENTS	No events
WD_INVALID_PARAMETER	Invalid parameter
WD_INCORRECT_VERSION	Incorrect WinDriver version installed
WD_TRY_AGAIN	Try again
WD_INVALID_IOCTL	Received an invalid IOCTL

A.7.3 Status Codes Returned by USBD

The following WinDriver status codes comply with USBD_XXX status codes returned by the USB stack drivers.

Status Code	**Description**
USBD Status Types	
WD_USBD_STATUS_SUCCESS	USBD: Success
WD_USBD_STATUS_PENDING	USBD: Operation pending
WD_USBD_STATUS_ERROR	USBD: Error
WD_USBD_STATUS_HALTED	USBD: Halted
USBD Status Codes (NOTE: These are comprised of one of the status types above and an error code, i.e., 0xXYYYYYYYL, where X=status type and YYYYYYY=error code. The same error codes may also appear with one of the other status types as well.)	
HC (Host Controller) Status Codes (NOTE: These use the WD_USBD_STATUS_HALTED status type.)	
WD_USBD_STATUS_CRC	HC status: CRC
WD_USBD_STATUS_BTSTUFF	HC status: Bit stuffing
WD_USBD_STATUS_DATA_TOGGLE_MISMATCH	HC status: Data toggle mismatch
WD_USBD_STATUS_STALL_PID	HC status: PID stall
WD_USBD_STATUS_DEV_NOT_RESPONDING	HC status: Device not responding
WD_USBD_STATUS_PID_CHECK_FAILURE	HC status: PID check failed
WD_USBD_STATUS_UNEXPECTED_PID	HC status: Unexpected PID
WD_USBD_STATUS_DATA_OVERRUN	HC status: Data overrun
WD_USBD_STATUS_DATA_UNDERRUN	HC status: Data underrun
WD_USBD_STATUS_RESERVED1	HC status: Reserved1
WD_USBD_STATUS_RESERVED2	HC status: Reserved2
WD_USBD_STATUS_BUFFER_OVERRUN	HC status: Buffer overrun
WD_USBD_STATUS_BUFFER_UNDERRUN	HC status: Buffer Underrun
WD_USBD_STATUS_NOT_ACCESSED	HC status: Not accessed
WD_USBD_STATUS_FIFO	HC status: FIFO

Status Code	Description
For Windows only:	
WD_USBD_STATUS_XACT_ERROR	HC status: The host controller has set the Transaction Error (XactErr) bit in the transfer descriptor's status field
WD_USBD_STATUS_BABBLE_DETECTED	HC status: Babble detected
WD_USBD_STATUS_DATA_BUFFER_ERROR	HC status: Data buffer error
For Windows CE only:	
WD_USBD_STATUS_NOT_COMPLETE	USBD: Transfer not completed
WD_USBD_STATUS_CLIENT_BUFFER	USBD: Cannot write to buffer
For all platforms:	
WD_USBD_STATUS_CANCELED	USBD: Transfer cancelled
Returned by HCD (Host Controller Driver) if a transfer is submitted to an endpoint that is stalled:	
WD_USBD_STATUS_ENDPOINT_HALTED	HCD: Transfer submitted to stalled endpoint
Software Status Codes (NOTE: Only the error bit is set):	
WD_USBD_STATUS_NO_MEMORY	USBD: Out of memory
WD_USBD_STATUS_INVALID_URB_FUNCTION	USBD: Invalid URB function
WD_USBD_STATUS_INVALID_PARAMETER	USBD: Invalid parameter
Returned if client driver attempts to close an endpoint/interface or configuration with outstanding transfers:	
WD_USBD_STATUS_ERROR_BUSY	USBD: Attempted to close endpoint/interface/configuration with outstanding transfer
Returned by USBD if it cannot complete a URB request. Typically this will be returned in the URB status field (when the IRP is completed) with a more specific NT error code. The IRP status codes are indicated in WinDriver's Debug Monitor tool (wddebug_gui):	
WD_USBD_STATUS_REQUEST_FAILED	USBD: URB request failed
WD_USBD_STATUS_INVALID_PIPE_HANDLE	USBD: Invalid pipe handle
Returned when there is not enough bandwidth available to open a requested endpoint:	
WD_USBD_STATUS_NO_BANDWIDTH	USBD: Not enough bandwidth for endpoint
Generic HC (Host Controller) error:	
WD_USBD_STATUS_INTERNAL_HC_ERROR	USBD: Host controller error
Returned when a short packet terminates the transfer, i.e., USBD_SHORT_TRANSFER_OK bit not set:	
WD_USBD_STATUS_ERROR_SHORT_TRANSFER	USBD: Transfer terminated with short packet

Status Code	Description
Returned if the requested start frame is not within USBD_ISO_START_FRAME_RANGE of the current USB frame (NOTE: The stall bit is set):	
WD_USBD_STATUS_BAD_START_FRAME	USBD: Start frame outside range
Returned by HCD (Host Controller Driver) if all packets in an isochronous transfer complete with an error:	
WD_USBD_STATUS_ISOCH_REQUEST_FAILED	HCD: Isochronous transfer completed with error
Returned by USBD if the frame length control for a given HC (Host Controller) is already taken by another driver:	
WD_USBD_STATUS_FRAME_CONTROL_OWNED	USBD: Frame length control already taken
Returned by USBD if the caller does not own frame length control and attempts to release or modify the HC frame length:	
WD_USBD_STATUS_FRAME_CONTROL_NOT_OWNED	USBD: Attempted operation on frame length control not owned by caller
Additional software error codes added for USB 2.0 (for Windows only):	
WD_USBD_STATUS_NOT_SUPPORTED	USBD: API not supported/implemented
WD_USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR	USBD: Invalid configuration descriptor
WD_USBD_STATUS_INSUFFICIENT_RESOURCES	USBD: Insufficient resources
WD_USBD_STATUS_SET_CONFIG_FAILED	USBD: Set configuration failed
WD_USBD_STATUS_BUFFER_TOO_SMALL	USBD: Buffer too small
WD_USBD_STATUS_INTERFACE_NOT_FOUND	USBD: Interface not found
WD_USBD_STATUS_INAVLID_PIPE_FLAGS	USBD: Invalid pipe flags
WD_USBD_STATUS_TIMEOUT	USBD: Timeout
WD_USBD_STATUS_DEVICE_GONE	USBD: Device gone
WD_USBD_STATUS_STATUS_NOT_MAPPED	USBD: Status not mapped
Extended isochronous error codes returned by USBD. These errors appear in the packet status field of an isochronous transfer:	
WD_USBD_STATUS_ISO_NOT_ACCESSED_BY_HW	USBD: The controller did not access the TD associated with this packet
WD_USBD_STATUS_ISO_TD_ERROR	USBD: Controller reported an error in the TD
WD_USBD_STATUS_ISO_NA_LATE_USBPORT	USBD: The packet was submitted in time by the client but failed to reach the miniport in time

Status Code	Description
WD_USBD_STATUS_ISO_NOT_ACCESSED_LATE	USBD: The packet was not sent because the client submitted it too late to transmit

Appendix B

USB Device – Cypress EZ-USB FX2LP CY7C68013A API Reference

B.1 Firmware Library API

This section describes the WinDriver USB Device firmware library API for the Cypress EZ-USB FX2LP CY7C68013A development board. The functions and general types and definitions described in this section are declared and defined (respectively) in the **FX2LP\include\wdf_cypress_lib.h** header file. The functions are implemented in the generated DriverWizard **wdf_cypress_lib.c** file – for registered users, or in the **FX2LP\lib\wdf_cypress_fx2lp_eval.lib** evaluation firmware library – for evaluation users (see section 12.3.5 for details).

> **NOTE**
> Registered users can modify the library source code. When modifying the code, make sure that you comply with the USB Specification and with your hardware's specification – see note in section 12.4.3.

B.1.1 Firmware Library Types

The APIs described in this section are defined in **FX2LP\wdf_cypress_lib.h**.

B.1.1.1 EP_DIR Enumeration

Enumeration of endpoint directions:

Enum Value	**Description**
DIR_OUT	Direction OUT (write from the host to the device)
DIR_IN	Direction IN (read from the device to the host)

B.1.1.2 EP_TYPE Enumeration

Enumeration of endpoint types.
The endpoint's type determines the type of transfers to be performed on the endpoint – bulk, interrupt or isochronous.

Enum Value	**Description**
ISOCHRONOUS	Isochronous endpoint
BULK	Bulk endpoint
INTERRUPT	Interrupt endpoint

B.1.1.3 EP_BUFFERING Enumeration

Enumeration of endpoint buffering types:

Enum Value	**Description**
DOUBLE_BUFFERING	Double buffering
TRIPLE_BUFFERING	Triple buffering
QUAD_BUFFERING	Quadruple buffering

B.1.2 Firmware Library Functions

The functions described in this section are declared in **FX2LP\wdf_cypress_lib.h**.

B.1.2.1 WDF_EP1INConfig() / WDF_EP1OUTConfig()

PURPOSE

- Configures endpoint 1 for IN transfers (`WDF_EP1INConfig()`) or OUT transfers (`WDF_EPOUTConfig()`).

PROTOTYPE

```
void WDF_EP1INConfig(EP_TYPE type);
void WDF_EP1OUTConfig(EP_TYPE type);
```

PARAMETERS

Name	Type	Input/Output
➢ type	EP_TYPE	Input

DESCRIPTION

Name	Description
type	The endpoint's transfer type [B.1.1.2]

RETURN VALUE

None

B.1.2.2 WDF_EP2Config / WDF_EP6Config()

> **NOTE**
> The prototype and description of **`WDF_EP2Config()`** and **`WDF_EP6Config()`** is identical, except for the endpoint number. The description below will refer to endpoint 2, but you can simply replace all "2" references with "6" to get the description of `WDF_EP6Config()`.

PURPOSE

• Configures endpoint 2.

PROTOTYPE

```
void WDF_EP2Config(
    EP_DIR dir,
    EP_TYPE type,
    EP_BUFFERING buffering,
    int size,
    int nPacketPerMF);
```

PARAMETERS

Name	Type	Input/Output
➢ dir	EP_DIR	Input
➢ type	EP_TYPE	Input
➢ buffering	EP_BUFFERING	Input
➢ size	int	Input
➢ nPacketPerMF	int	Input

Description

Name	Description
dir	The endpoint's direction [B.1.1.1]
type	The endpoint's transfer type [B.1.1.2]
buffering	The endpoint's buffering type [B.1.1.3]
size	The size of the endpoint's FIFO buffer (in bytes)
nPacketPerMF	Number of packets per microframe

Return Value

None

B.1.2.3 WDF_EP4Config / WDF_EP8Config()

> **NOTE**
> The prototype and description of **`WDF_EP4Config()`** and **`WDF_EP8Config()`** is identical, except for the endpoint number. The description below will refer to endpoint 4, but you can simply replace all "4" references with "8" to get the description of `WDF_EP8Config()`.

PURPOSE

- Configures endpoint 4.

PROTOTYPE

```
void WDF_EP4Config(
    EP_DIR dir,
    EP_TYPE type);
```

PARAMETERS

Name	Type	Input/Output
➢ dir	EP_DIR	Input
➢ type	EP_TYPE	Input

DESCRIPTION

Name	Description
dir	The endpoint's direction [B.1.1.1]
type	The endpoint's transfer type [B.1.1.2]

RETURN VALUE

None

B.1.2.4 WDF_FIFOReset()

PURPOSE

• Restores an endpoint's FIFO (First In First Out) buffer to its default state.

PROTOTYPE

```
void WDF_FIFOReset(int ep);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	int	Input

DESCRIPTION

Name	Description
ep	Endpoint number (address)

RETURN VALUE

None

B.1.2.5 WDF_SkipOutPacket()

PURPOSE

• Signals an endpoint's FIFO (First In First Out) buffer to ignore received OUT packets.

PROTOTYPE

```
void WDF_SkipOutPacket(int ep);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	int	Input

DESCRIPTION

Name	Description
ep	Endpoint number (address)

RETURN VALUE

None

B.1.2.6 WDF_FIFOWrite()

PURPOSE

• Writes data to an endpoint's FIFO (First In First Out) buffer.
The call to this function should be followed by a call to
`WDF_SetEPByteCount()` [B.1.2.10].

PROTOTYPE

```
void WDF_FIFOWrite(
    int ep,
    BYTE buf[],
    int size);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	int	Input
➢ buf	BYTE[ ]	Input
➢ size	int	Input

DESCRIPTION

Name	Description
ep	Endpoint number (address)
buf	Data buffer to write
size	Number of bytes to write

RETURN VALUE

None

B.1.2.7 WDF_FIFORead()

Purpose

• Reads data from an endpoint's FIFO (First In First Out) buffer.
The call to this function should be preceded by a call to `WDF_GetEPByteCount()` [B.1.2.11] in order to determine the amount of bytes to read.

Prototype

```
void WDF_FIFORead(
    int ep,
    BYTE buf[],
    int size);
```

Parameters

Name	Type	Input/Output
➢ ep	int	Input
➢ buf	BYTE[ ]	Output
➢ size	int	Input

Description

Name	Description
ep	Endpoint number (address)
buf	Buffer to hold the read data
size	Number of bytes to read from the FIFO buffer

Return Value

None

B.1.2.8 WDF_FIFOFull()

PURPOSE

• Checks if an endpoint's FIFO (First In First Out) buffer is completely full.

PROTOTYPE

```
BOOL WDF_FIFOFull(int ep);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	int	Input

DESCRIPTION

Name	Description
ep	Endpoint number (address)

RETURN VALUE

Returns TRUE if the endpoint's FIFO buffer is full; otherwise returns FALSE.

B.1.2.9 WDF_FIFOEmpty()

PURPOSE

• Checks if an endpoint's FIFO (First In First Out) buffer is empty.

PROTOTYPE

```
BOOL WDF_FIFOEmpty(int ep);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	int	Input

DESCRIPTION

Name	Description
ep	Endpoint number (address)

RETURN VALUE

Returns TRUE if the endpoint's FIFO buffer is empty; otherwise returns FALSE.

B.1.2.10 WDF_SetEPByteCount()

PURPOSE

• Sets the bytes count of an endpoint's FIFO (First In First Out) buffer.
The call to this function should be preceded by a call to `WDF_FIFOWrite()` [B.1.2.6] in order to update the endpoint's FIFO buffer with the data to be transferred to the host.

PROTOTYPE

```
void WDF_SetEPByteCount(
    int ep,
    WORD bytes_count);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	int	Input
➢ bytes_count	WORD	Input

DESCRIPTION

Name	Description
ep	Endpoint number (address)
bytes_count	Bytes count to set

RETURN VALUE

None

B.1.2.11 WDF_GetEPByteCount()

PURPOSE

• Gets the current bytes count of an endpoint's FIFO (First In First Out) buffer.
This function should be called before calling `WDF_FIFORead()` [B.1.2.7] to read from the endpoint's FIFO buffer, in order to determine the amount of bytes to read.

PROTOTYPE

```
WORD WDF_GetEPByteCount(int ep);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	int	Input

DESCRIPTION

Name	Description
ep	Endpoint number (address)

RETURN VALUE

Returns the endpoint's FIFO bytes count.

B.1.2.12 WDF_I2CInit()

PURPOSE

• Initializes the I2C bus.

PROTOTYPE

```
void WDF_I2CInit(void);
```

RETURN VALUE

None

B.1.2.13 WDF_SetDigitLed()

PURPOSE

• Displays the specified digit in the development board's digit LED.

PROTOTYPE

```
void WDF_SetDigitLed(int digit);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ digit	int	Input

DESCRIPTION

Name	**Description**
digit	The digit to display

RETURN VALUE

None

B.1.2.14 WDF_I2CWrite()

PURPOSE

• Writes data to a specified address on the I2C bus.

PROTOTYPE

```
BOOL WDF_I2CWrite(
    BYTE addr,
    BYTE len,
    BYTE xdata *dat);
```

PARAMETERS

Name	Type	Input/Output
➢ addr	BYTE	Input
➢ len	BYTE	Input
➢ dat	xdata*	Input

DESCRIPTION

Name	Description
addr	The address to which to write
len	The number of bytes to write
dat	Pointer to a buffer containing the data to write

RETURN VALUE

Returns TRUE for a successful write operation; otherwise returns FALSE.

B.1.2.15 WDF_I2CRead()

PURPOSE

- Reads data from a specified address on the I2C bus.

PROTOTYPE

```
BOOL WDF_I2CRead(
    BYTE addr,
    BYTE len,
    BYTE xdata *dat);
```

PARAMETERS

Name	Type	Input/Output
➢ addr	BYTE	Input
➢ len	BYTE	Input
➢ dat	xdata*	Output

DESCRIPTION

Name	Description
addr	The address from which to read
len	The number of bytes to read
dat	Pointer to a buffer containing the data that is read

RETURN VALUE

Returns TRUE for a successful read operation; otherwise returns FALSE.

B.1.2.16 WDF_I2CWaitForEEPROMWrite()

PURPOSE

• Waits for the completion of the current write operation on the specified I2C bus address.

PROTOTYPE

```
void WDF_I2CWaitForEEPROMWrite(BYTE addr);
```

PARAMETERS

Name	Type	Input/Output
➢ addr	BYTE	Input

DESCRIPTION

Name	Description
addr	The I2C bus address on which to wait

RETURN VALUE

None

B.1.2.17 WDF_I2CGetStatus()

PURPOSE

• Gets the current status of the I2C bus.

PROTOTYPE

```
int WDF_I2CGetStatus(void);
```

RETURN VALUE

Returns the I2C bus status.

B.1.2.18 WDF_I2CClearStatus()

PURPOSE

• Clears the I2C bus status from errors/NAKs.

PROTOTYPE

```
void WDF_I2CClearStatus(void);
```

RETURN VALUE

None

B.2 Generated DriverWizard Firmware API

This section describes the WinDriver USB Device generated DriverWizard firmware API for the Cypress EZ-USB FX2LP CY7C68013A development board. The functions described in this section are declared in the **FX2LP\include\periph.h** header file and implemented in the generated DriverWizard **periph.c** file, according to the device configuration information defined in the wizard.

The firmware's entry point – `main()` in **main.c** (source code provided for registered users only) – implements a **Task Dispatcher**, which calls the `WDF_xxx()` functions declared in **periph.h** (and implemented in **periph.c**) in order to communicate with the peripheral device.

NOTE
When modifying the generated code, make sure that you comply with the USB Specification and with your hardware's specification – see note in section [12.4.3].

B.2.1 WDF_Init()

PURPOSE

• Initializes the device.
This function is automatically called from the firmware's `main()` function in order to perform the required initialization to enable communication with the device.

PROTOTYPE

```
void WDF_Init(void);
```

RETURN VALUE

None

B.2.2 WDF_Poll()

PURPOSE

• Polls the device for FIFO data.
The Task Dispatcher calls this function repeatedly while the device is idle.

PROTOTYPE

```
void WDF_Poll(void);
```

RETURN VALUE

None

B.2.3 WDF_Suspend()

PURPOSE

• This function is called by the Task Dispatcher before the device goes into suspend mode.

PROTOTYPE

```
BOOL WDF_Suspend(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.4 WDF_Resume()

PURPOSE

• This function is called by the Task Dispatcher after the device resumes activity.

PROTOTYPE

```
BOOL WDF_Resume(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.5 WDF_GetDescriptor()

PURPOSE

• This function is called by the Task Dispatcher when a GET DESCRIPTOR command is received.

PROTOTYPE

```
BOOL WDF_GetDescriptor(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.6 WDF_SetConfiguration()

PURPOSE

• This function is called by the Task Dispatcher when a SET CONFIGURATION command is received.

PROTOTYPE

```
BOOL WDF_SetConfiguration(BYTE config);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ config	BYTE	Input

DESCRIPTION

Name	**Description**
config	Configuration number to set

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.7 WDF_GetConfiguration()

PURPOSE

• This function is called by the Task Dispatcher when a GET CONFIGURATION command is received.

PROTOTYPE

```
BOOL WDF_GetConfiguration(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.8 WDF_SetInterface()

PURPOSE

• This function is called by the Task Dispatcher when a SET INTERFACE command is received.

PROTOTYPE

```
BOOL WDF_SetInterface(
    BYTE ifc,
    BYTE alt_set);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ ifc	BYTE	Input
➢ alt_set	BYTE	Input

DESCRIPTION

Name	**Description**
ifc	Interface number to set
alt_set	Alternate setting number to set

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.9 WDF_GetInterface()

PURPOSE

• This function is called by the Task Dispatcher when a GET INTERFACE command is received.

PROTOTYPE

```
BOOL WDF_GetInterface(BYTE ifc);
```

PARAMETERS

Name	Type	Input/Output
➢ ifc	BYTE	Input

DESCRIPTION

Name	Description
ifc	Interface number

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.10 WDF_GetStatus()

PURPOSE

• This function is called by the Task Dispatcher when a GET STATUS command is received.

PROTOTYPE

```
BOOL WDF_GetStatus(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.11 WDF_ClearFeature()

PURPOSE

• This function is called by the Task Dispatcher when a CLEAR FEATURE command is received.

PROTOTYPE

```
BOOL WDF_ClearFeature(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.12 WDF_SetFeature()

PURPOSE

• This function is called by the Task Dispatcher when a SET FEATURE command is received.

PROTOTYPE

```
BOOL WDF_SetFeature(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

B.2.13 WDF_VendorCmnd()

PURPOSE

• This function is called by the Task Dispatcher when a vendor-specific command is received.

PROTOTYPE

```
BOOL WDF_VendorCmnd(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

Appendix C

USB Device – Microchip PIC18F4550 API Reference

C.1 Firmware Library API

This section describes the WinDriver USB Device standard firmware library API for the Microchip PIC18F4550 development board. The functions, macros and general types and definitions described in this section are declared and defined (respectively) in the **18F4550\include\wdf_microchip_lib.h**, **18F4550\include\types.h** and **18F4550\include\wdf_usb9.h** header files. The functions are implemented in the generated DriverWizard **wdf_microchip_lib.c** and **wdf_usb9.c** files – for registered users, or in the **18F4550\lib\wdf_microchip_18f4550_eval.lib** evaluation firmware library – for evaluation users (see section 12.3.5 for details).

The Microchip PIC18F4550 board's **mass storage** firmware library contains the same firmware files and APIs that comprise the standard firmware library, as described in this section. In addition, the mass storage library defines mass storage specific APIs, which are described in section [C.2] below.

> **NOTE**
> Registered users can modify the library source code. When modifying the code, make sure that you comply with the USB Specification and with your hardware's specification – see note in section 12.4.3.

C.1.1 Firmware Library Types

The data types described in this section are defined in the **18F4550\include\types.h** header file.

C.1.1.1 EP_DIR Enumeration

Enumeration of endpoint directions:

Enum Value	**Description**
OUT	Direction OUT (write from the host to the device)
IN	Direction IN (read from the device to the host)

C.1.1.2 EP_TYPE Enumeration

Enumeration of endpoint types.
The endpoint's type determines the type of transfers to be performed on the endpoint – bulk, interrupt or isochronous.

Enum Value	**Description**
ISOCHRONOUS	Isochronous endpoint
BULK	Bulk endpoint
INTERRUPT	Interrupt endpoint
CONTROL	Control endpoint

C.1.1.3 WDF_TRIGGER_OPTIONS Enumeration

Trigger options enumeration:

Enum Value	**Description**
TRIGGER_NO_TOGGLE_DTS	Do not toggle the Data Toggle Synchronization bit (DTS)
TRIGGER_IGNORE_DTS	Ignore the value of the DTS
TRIGGER_DAT0	Toggle the DTS to DAT0
TRIGGER_DAT1	Toggle the DTS to DAT1

C.1.1.4 BD_STAT Union

Endpoint buffer descriptor status union type:

Name	Type	Description
➢ _byte	byte	
➢	struct	
❏ BC8	bit field (1)	Bit 8 of the endpoint's last transfer byte count
❏ BC9	bit field (1)	Bit 9 (MSB) of the endpoint's last transfer byte count
❏ BSTALL	bit field (1)	Buffer stall enable
❏ DTSEN	bit field (1)	Data toggle synchronization enable
❏ INCDIS	bit field (1)	Address increment disable
❏ KEN	bit field (1)	Buffer descriptor keep enable
❏ DTS	bit field (1)	Data toggle synchronization value
❏ UOWN	bit field (1)	USB ownership
➢	struct	
❏ BC8	bit field (1)	Bit 8 of the endpoint's last transfer byte count
❏ BC9	bit field (1)	Bit 9 (MSB) of the endpoint's last transfer byte count
❏ PID0	bit field (1)	Bit 0 of the packet identifier
❏ PID1	bit field (1)	Bit 1 of the packet identifier
❏ PID2	bit field (1)	Bit 2 of the packet identifier
❏ PID3	bit field (1)	Bit 3 of the packet identifier
❏	bit field (1)	Reserved
❏ UOWN	bit field (1)	USB ownership
➢	struct	
❏	bit field (2)	Reserved
❏ PID	bit field (4)	Packet identifier
❏	bit field (2)	Reserved

C.1.1.5 BDT Union

Endpoint buffer descriptor table union type:

Name	Type	Description
➢	struct	
❏ Stat	BD_STAT	Buffer descriptor status [C.1.1.4]
❏ Cnt	byte	The endpoint's last transfer byte count. The byte count's most significant bits are stored in the `BC8` and `BC9` fields of the `BD_STAT` union (**`Stat`**)
❏ ADRL	byte	Low buffer address
❏ ADRH	byte	High buffer address
➢	struct	
❏	bit field (8)	Reserved
❏	bit field (8)	Reserved
❏ ADR	byte*	Pointer to the buffer address

C.1.1.6 WORD Union

2-bytes access union type:

Name	Type	Description
➢ _word	word	`word` type
➢	struct	2-bytes array struct
❏ v	byte[2]	2-bytes array

C.1.1.7 DWORD Union

4-bytes access union type:

Name	Type	Description
➢ _dword	dword	`dword` type
➢	struct	4 bytes struct
❏ byte0	byte	1st byte
❏ byte1	byte	2nd byte
❏ byte2	byte	3rd byte
❏ byte3	byte	4th byte
➢	struct	2 words struct
❏ word0	word	1st word
❏ word1	word	2nd word
➢	struct	2 WORDs struct
❏ Word0	WORD	1st WORD [C.1.1.6]
❏ Word1	WORD	2nd WORD [C.1.1.6]
➢	struct	4-bytes array struct
❏ v	byte[4]	4-bytes array

C.1.1.8 POINTER Union

Pointer union type:

Name	Type	Description
➢ _pFunc	typedef void(*pFunc)(void);	Function pointer
➢ bRam	byte*	RAM byte pointer: a 2-bytes pointer pointing to 1 byte of data
➢ wRam	word*	RAM word pointer: a 2-bytes pointer pointing to 2 bytes of data
➢ bRom	byte*	ROM byte pointer; size depends on compiler settings
➢ wRom	word*	ROM word pointer; size depends on compiler settings

C.1.1.9 USB_DEVICE_STATUS Union

USB device status union type:

Name	Type	Description
➢ _byte	byte	Device status information
➢	struct	Status information structure:
❏ remote_wakeup	bit field (1)	Device remote wake-up status: • 0: Remote wake-up disabled • 1: Remote wake-up enabled
❏ self_powered	bit field (1)	Device self-powered status: • 0: Device is bus powered • 1: Device is self powered
❏ ctr_trf_mem	bit field (1)	Location of data for the active control transfer: • 0: RAM • 1: ROM

C.1.1.10 CTRL_TRF_SETUP Union

Control transfer setup packet information union type:

Name	Type	Description
➢	struct	
❏ bmRequestType	byte	Request Type: *Bit 7*: Request direction (0=Host to device - Out, 1=Device to host - In). *Bits 5-6*: Request type (0=standard, 1=class, 2=vendor, 3=reserved). *Bits 0-4*: Recipient (0=device, 1=interface, 2=endpoint,3=other).
❏ bRequest	byte	The specific request
❏ wValue	word	A word-size value that varies according to the request. This value is typically used to specify an endpoint or an interface.
❏ wIndex	word	A word-size value that varies according to the request
❏ wLength	word	The length (in bytes) of the request's data segment – i.e. the number of bytes to transfer if there is a data stage

Name	Type	Description
➢	struct	
❑	bit field (8)	
❑	bit field (8)	
❑W_Value	WORD	
❑W_Index	WORD	
❑W_Length	WORD	
➢	struct	
❑Recipient	bit field (5)	The request's recipient: device / interface / endpoint / other
❑RequestType	bit field (2)	Request type: standard / class / vendor / reserved
❑DataDir	bit field (1)	The transfer's direction: • 0 – host to device (OUT) • 1 – device to host (IN)
❑	bit field (8)	
❑bFeature	byte	
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
➢	struct	
❑	bit field (8)	
❑	bit field (8)	
❑bDscIndex	byte	Descriptor index (relevant only for configuration and string descriptors)
❑bDescType	byte	Descriptor type – device / configuration / string
❑wLangID	word	Language ID (relevant for string descriptors)
❑	bit field (8)	
❑bFeature	byte	
❑	bit field (8)	
➢	struct	
❑	bit field (8)	
❑	bit field (8)	
❑bDevADR	byte	Device address (0 – 127)
❑bDevADRH	byte	Must be set to zero
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	

Name	Type	Description
➢	struct	
❑	bit field (8)	
❑	bit field (8)	
❑ bCfgValue	byte	Configuration number (0 – 255)
❑ bCfgRSD	byte	Must be set to zero (reserved)
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
➢	struct	
❑	bit field (8)	
❑	bit field (8)	
❑ bAltID	byte	Alternate setting number (0 – 255)
❑ bAltID_H	byte	Must be set to zero
❑ bIntfID	byte	Interface number (0 – 255)
❑ bIntfID_H	byte	Must be set to zero
❑	bit field (8)	
❑	bit field (8)	
➢	struct	
❑	bit field (8)	
❑	bit field (8)	
❑ bEPID	byte	Endpoint ID – number and direction
❑ bEPID_H	byte	Must be set to zero
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
➢	struct	
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	
❑ EPNum	bit field (4)	Endpoint number (0 – 15)
❑	bit field (3)	
❑ EPDir	bit field (1)	Endpoint direction: • 0 – OUT • 1 – IN
❑	bit field (8)	
❑	bit field (8)	
❑	bit field (8)	

C.1.1.11 EP_DATA Structure

Endpoint data structure type:

Name	Type	Description
number	byte	Endpoint number
reg	near byte*	UEPn register address
max_packet	word	The endpoint's maximum packet size (in bytes)
e_bdt	BDT*	Pointer to the endpoint's even buffer descriptor table [C.1.1.5]
o_bdt	BDT*	Pointer to the endpoint's odd buffer descriptor table [C.1.1.5]
e_buffer	byte*	Pointer to the endpoint's even data buffer
o_buffer	byte*	Pointer to the endpoint's odd data buffer

C.1.1.12 USB_DEVICE_CTX Structure

USB device context data structure type:

Name	Type	Description
ep0_out	EP_DATA	Endpoint data structure [C.1.1.11] for control pipe (endpoint 0) OUT requests
ep0_in	EP_DATA	Endpoint data structure [C.1.1.11] for control pipe (endpoint 0) IN requests
pSrc	POINTER	Data source pointer [C.1.1.8] for the active control transfer
pDst	POINTER	Data destination pointer [C.1.1.8] for the active control transfer
wCount	WORD	The size of the data (in bytes) for the data stage of the active control transfer
usb_stat	USB_DEVICE_STATUS	USB device status information [C.1.1.9]
usb_device_state	byte	Device state
usb_active_cfg	byte	The number of the active device configuration
ctrl_trf_state	byte	The state of the active control transfer

C.1.2 wdf_microchip_lib.h Functions and Macros

This section describes the firmware library's general functions and macros.

The functions and macros described in this section are declared and defined (respectively) in the **wdf_microchip_lib.h** header file. The functions are implemented in the generated DriverWizard **wdf_microchip_lib.c** file – for registered users, or in the **18F4550\lib\wdf_microchip_18f4550_eval.lib** evaluation firmware library – for evaluation users (see section 12.3.5 for details).

C.1.2.1 WDF_EPConfig()

PURPOSE

- Configures and enables a given endpoint for USB transfers.

PROTOTYPE

```
void WDF_EPConfig(
    EP_DATA *ep_data,
    byte ep_num,
    EP_DIR dir,
    EP_TYPE type,
    word max_packet,
    near byte *reg,
    BDT *e_bdt,
    byte *e_buffer,
    BDT *o_bdt,
    byte *o_buffer);
```

Parameters

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input/Output
➢ ep_num	byte	Input
➢ dir	EP_DIR	Input
➢ type	EP_TYPE	Input
➢ max_packet	word	Input
➢ reg	near byte*	Input
➢ e_bdt	BDT*	Input
➢ e_buffer	byte*	Input
➢ o_bdt	BDT*	Input
➢ o_buffer	byte*	Input

Description

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]
ep_num	The endpoint's number
dir	The endpoint's direction [C.1.1.1]
type	The endpoint's transfer type [C.1.1.2]
max_packet	The endpoint's maximum packet size (in bytes)
reg	Pointer to the endpoint's UEPn register
e_bdt	Pointer to the endpoint's even buffer descriptor table [C.1.1.5]
e_buffer	Pointer to the endpoint's even data buffer
o_bdt	Pointer to the endpoint's odd buffer descriptor table [C.1.1.5]. Can be NULL when double buffering is disabled.
o_buffer	Pointer to the endpoint's odd data buffer. Can be NULL when double buffering is disabled.

Return Value

None

C.1.2.2 WDF_EPWrite()

PURPOSE

• Writes data from a random-access memory (RAM) buffer to a given endpoint.

Should be followed by a call to `WDF_TriggerWriteTransfer()` [C.1.2.9] or `WDF_TriggerOptionWriteTransfer()` [C.1.2.10].

PROTOTYPE

```
void WDF_EPWrite(
    EP_DATA *ep_data,
    byte *buffer,
    word len);
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input
➢ buffer	byte*	Input
➢ len	word	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]
buffer	Pointer to a buffer containing the data to write
len	The number of bytes to write

RETURN VALUE

None

C.1.2.3 WDF_EPWriteRom()

PURPOSE

• Writes data from a read-only memory (ROM) buffer to a given endpoint.

Should be followed by a call to `WDF_TriggerWriteTransfer()` [C.1.2.9] or `WDF_TriggerOptionWriteTransfer()` [C.1.2.10].

PROTOTYPE

```
void WDF_EPWriteRom(
    EP_DATA *ep_data,
    rom byte *buffer,
    word len);
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input
➢ buffer	byte*	Input
➢ len	word	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]
buffer	Pointer to a ROM buffer containing the data to write
len	The number of bytes to write

RETURN VALUE

None

C.1.2.4 WDF_EPWriteNoCopy()

PURPOSE

• Writes data to a given endpoint using the input RAM buffer.
The data is written directly from the function's input data buffer address (**`addr`**), without being copied to the data buffer contained in the endpoint's data structure (**`ep_data`**). NOTE: The input data **`buffer`** in the call to this function must be within the 0x400 – 0x7FF 1KB RAM address space.

Should be followed by a call to `WDF_TriggerWriteTransfer()` [C.1.2.9] or `WDF_TriggerOptionWriteTransfer()` [C.1.2.10].

PROTOTYPE

```
void WDF_EPWriteNoCopy(
    EP_DATA *ep_data,
    byte *buffer,
    word len);
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input
➢ buffer	rom byte*	Input
➢ len	word	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]
buffer	Pointer to a buffer containing the data to write
len	The number of bytes to write

RETURN VALUE

None

C.1.2.5 WDF_EPRead()

PURPOSE

• Reads data from a given endpoint to a random-access memory (RAM) buffer.

Should be preceded by a call to `WDF_TriggerReadTransfer()` [C.1.2.11] or `WDF_TriggerOptionReadTransfer()` [C.1.2.12].

PROTOTYPE

```
word WDF_EPRead(
    EP_DATA *ep_data,
    byte *buffer,
    word len);
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input
➢ buffer	byte*	Output
➢ len	word	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]
buffer	Pointer to a buffer to be updated with the read data
len	The number of bytes to read

RETURN VALUE

Returns the number of bytes that were read.

C.1.2.6 WDF_IsEPStall()

PURPOSE

• Checks if the given endpoint is currently stalled.

PROTOTYPE

```
BOOL WDF_IsEPStall(EP_DATA *ep_data);
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]

RETURN VALUE

Returns TRUE if the endpoint is currently stalled; otherwise returns FALSE.

C.1.2.7 WDF_IsEPBusy()

PURPOSE

- Checks if the given endpoint is currently busy.

PROTOTYPE

```
WDF_IsEPBusy( ep_data )
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]

RETURN VALUE

Returns TRUE if the endpoint is currently busy; otherwise returns FALSE.

C.1.2.8 WDF_IsEPDataReady()

PURPOSE

• Checks if the given endpoint contains data received from the host.

PROTOTYPE

```
WDF_IsEPDataReady( ep_data )
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]

RETURN VALUE

Returns TRUE if the endpoint contains data from the host; otherwise returns FALSE.

C.1.2.9 WDF_TriggerWriteTransfer()

PURPOSE

• Triggers a write data transfer on a given endpoint, transferring the USB ownership of the relevant buffer descriptor to the SIE.

PROTOTYPE

```
WDF_TriggerWriteTransfer(ep_data)
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]

RETURN VALUE

None

C.1.2.10 WDF_TriggerOptionWriteTransfer()

PURPOSE

• Triggers a write data transfer on a given endpoint, transferring the USB ownership of the relevant buffer descriptor to the SIE.
This function enables the user to set the trigger options [C.1.1.3] for the write transfer.

PROTOTYPE

```
void WDF_TriggerOptionWriteTransfer(
    EP_DATA *ep_data,
    byte options);
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input
➢ options	byte	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]
options	Trigger option – can be set to zero (no option) or to any of the `WDF_TRIGGER_OPTIONS` enumeration values [C.1.1.3]

RETURN VALUE

None

C.1.2.11 WDF_TriggerReadTransfer()

PURPOSE

• Triggers a read data transfer on a given endpoint, transferring the USB ownership of the relevant buffer descriptor to the SIE.

PROTOTYPE

```
WDF_TriggerReadTransfer(ep_data)
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]

RETURN VALUE

None

C.1.2.12 WDF_TriggerOptionReadTransfer()

PURPOSE

• Triggers a read data transfer on a given endpoint, transferring the USB ownership of the relevant buffer descriptor to the SIE.
This function enables the user to set the trigger options [C.1.1.3] for the read transfer.

PROTOTYPE

```
void WDF_TriggerOptionReadTransfer(
    EP_DATA *ep_data,
    byte options);
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input
➢ options	byte	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]
options	Trigger option – can be set to zero (no option) or to any of the `WDF_TRIGGER_OPTIONS` enumeration values [C.1.1.3]

RETURN VALUE

None

C.1.2.13 WDF_TriggerReadTransferNoCopy()

PURPOSE

• Triggers a read data transfer on a given endpoint, transferring the USB ownership of the relevant buffer descriptor to the SIE.
The read data will be copied to the function's input data buffer address (**`addr`**) instead of to the data buffer contained in the endpoint's data structure (**`ep_data`**).
NOTE: The address of the function's input data buffer (**`addr`**) must be within the 0x400 – 0x7FF 1KB RAM address space.

After the call to this function the read data will be available in the caller's data buffer (**`addr`**), therefore you should **not** follow the call to this function with a call to `WDF_EPRead()` [C.1.2.5].

PROTOTYPE

```
void WDF_TriggerReadTransferNoCopy(
    EP_DATA *ep_data,
    byte *addr);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ ep_data	EP_DATA*	Input
➢ addr	byte*	Input

DESCRIPTION

Name	**Description**
ep_data	Pointer to an endpoint data structure [C.1.1.11]
addr	Pointer to the beginning of a data buffer to which the read data should be copied

RETURN VALUE

None

C.1.2.14 WDF_GetReadBytesCount()

PURPOSE

• Gets the current bytes count in a given endpoint's read buffer.
This function should be called before calling `WDF_EPRead()` [C.1.2.5] to read from the endpoint, in order to determine the amount of bytes to read.

PROTOTYPE

```
WORD WDF_GetReadBytesCount(EP_DATA *ep_data);
```

PARAMETERS

Name	Type	Input/Output
➢ ep_data	EP_DATA*	Input

DESCRIPTION

Name	Description
ep_data	Pointer to an endpoint data structure [C.1.1.11]

RETURN VALUE

Returns the endpoint's read buffer bytes count.

C.1.2.15 WDF_DisableEP1to15()

PURPOSE

• Disables endpoints 1 to 15.

PROTOTYPE

```
void WDF_DisableEP1to15(void);
```

RETURN VALUE

None

C.1.2.16 WDF_DisableEP()

PURPOSE

• Disables a given endpoint.

PROTOTYPE

```
void WDF_DisableEP(EP_DATA *ep);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ ep_data	EP_DATA*	Input

DESCRIPTION

Name	**Description**
ep_data	Pointer to an endpoint data structure [C.1.1.11]

RETURN VALUE

None

C.1.3 wdf_usb9.h Functions

This section describes the firmware library's USB descriptors functions, which support Chapter 9 of the USB 2.0 Specification.

The functions described in this section are declared in the **wdf_usb9.h** header file and implemented in the generated DriverWizard **wdf_usb9.c** file – for registered users, or in the **18F4550\lib\wdf_microchip_18f4550_eval.lib** evaluation firmware library – for evaluation users (see section 12.3.5 for details).

C.1.3.1 WDF_USBCheckStdRequest()

PURPOSE

• Verifies that a given standard USB control request complies with the USB 2.0 Specification, and if so, handles the request.

PROTOTYPE

```
BOOL USBCheckStdRequest(
    USB_DEVICE_CTX *device_ctx,
    CTRL_TRF_SETUP *setup,
    byte *data_buffer);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ device_ctx	USB_DEVICE_CTX*	Input/Output
➢ setup	CTRL_TRF_SETUP*	Input
➢ data_buffer	byte*	Output

DESCRIPTION

Name	**Description**
device_ctx	Pointer to a device context information structure [C.1.1.12]
setup	Pointer to a control transfer setup packet information union [C.1.1.10]
data_buffer	Pointer to the data buffer for the transfer

RETURN VALUE

Returns TRUE if the request was valid; otherwise returns FALSE.

C.2 Mass Storage Firmware Library API

This section describes the mass storage APIs of the WinDriver USB Device mass storage firmware library. The functions described in this section are declared in the **18F4550\include\class\msd\wdf_msd.h** header file and implemented in the generated DriverWizard **wdf_msd.c** file – for registered users, or in the **18F4550\lib\wdf_microchip_msd_18f4550_eval.lib** evaluation firmware library – for evaluation users (see section 12.3.5 for details).

> **NOTE**
> Registered users can modify the library source code. When modifying the code, make sure that you comply with the USB and Mass Storage Specifications and with your hardware's specification – see note in section 12.4.3.

C.2.1 WDF_MSD_Init()

PURPOSE

- Initializes a mass storage device.

PROTOTYPE

```
void WDF_MSD_Init(
    EP_DATA *ep_in,
    EP_DATA *ep_out,
    byte max_lun,
    byte interface,
    byte alternate_setting);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ ep_data_in	EP_DATA*	Input
➢ ep_data_out	EP_DATA*	Input
➢ max_lun	byte	Input
➢ interface	byte	Input
➢ alternate_setting	byte	Input

DESCRIPTION

Name	**Description**
ep_data_in	Pointer to the device's mass storage IN endpoint's data structure [C.1.1.11]
ep_data_out	Pointer to the device's mass storage OUT endpoint's data structure [C.1.1.11]
max_lun	The number of the last supported logical unit on the device (0 based).
interface	The number of the device's mass storage interface
alternate_setting	The number of the device's mass storage alternate setting

RETURN VALUE

None

C.2.2 WDF_MSD_USBCheckMSDRequest()

PURPOSE

• Verifies that a given USB control request complies with the USB Mass Storage Class Specification, and if so, handles the request.

PROTOTYPE

```
BOOL WDF_MSD_USBCheckMSDRequest(
    USB_DEVICE_CTX *device_ctx,
    CTRL_TRF_SETUP *setup,
    byte *data_buffer);
```

PARAMETERS

Name	Type	Input/Output
➢ device_ctx	USB_DEVICE_CTX*	Input/Output
➢ setup	CTRL_TRF_SETUP*	Input
➢ data_buffer	byte*	Output

DESCRIPTION

Name	Description
device_ctx	Pointer to a device context information structure [C.1.1.12]
setup	Pointer to a control transfer setup packet information union [C.1.1.10]
data_buffer	Pointer to the data buffer for the transfer

RETURN VALUE

Returns TRUE if the request was valid; otherwise returns FALSE.

C.2.3 WDF_MSD_ProcessIO()

PURPOSE

• Processes a mass storage SCSI command.

PROTOTYPE

```
void WDF_MSD_ProcessIO(void);
```

RETURN VALUE

None

C.3 Generated DriverWizard Firmware API

This section describes the WinDriver USB Device generated DriverWizard firmware API for the Microchip PIC18F4550 development board. The functions described in this section are declared in the **18F4550\include\periph.h** header file and implemented in the generated DriverWizard **periph.c** file, according to the device configuration information defined in the wizard.

The firmware's entry point – `main()` in **main.c** (source code provided for registered users only) – implements a **Task Dispatcher**, which calls the `WDF_xxx()` functions declared in **periph.h** (and implemented in **periph.c**) in order to communicate with the peripheral device.

The Microchip PIC18F4550 board's generated **mass storage** code includes, in addition to the files and APIs described in this section, specific mass storage APIs, which are described in section [C.4] below.

> **NOTE**
> When modifying the generated code, make sure that you comply with the USB Specification and with your hardware's specification – see note in section [12.4.3].

C.3.1 WDF_Init()

PURPOSE

• Initializes the device.
This function is automatically called from the firmware's `main()` function in order to perform the required initialization to enable communication with the device.

PROTOTYPE

```
void WDF_Init(void);
```

RETURN VALUE

None

C.3.2 WDF_Poll()

PURPOSE

• Polls the device for FIFO data.
The Task Dispatcher calls this function repeatedly while the device is idle.

PROTOTYPE

```
void WDF_Poll(void);
```

RETURN VALUE

None

C.3.3 WDF_SOFHandler()

PURPOSE

• Start of frame interrupt handler function.

PROTOTYPE

```
void WDF_SOFHandler(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

C.3.4 WDF_Suspend()

PURPOSE

• This function is called by the Task Dispatcher before the device goes into suspend mode.

PROTOTYPE

```
BOOL WDF_Suspend(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

C.3.5 WDF_Resume()

PURPOSE

• This function is called by the Task Dispatcher after the device resumes activity.

PROTOTYPE

```
BOOL WDF_Resume(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

C.3.6 WDF_ErrorHandler()

PURPOSE

- USB error interrupt handler function.

PROTOTYPE

```
void WDF_ErrorHandler(void);
```

RETURN VALUE

None

C.3.7 WDF_SetConfiguration()

PURPOSE

• This function is called by the Task Dispatcher when a SET CONFIGURATION command is received.

PROTOTYPE

```
void WDF_SetConfiguration(byte config);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ config	byte	Input

DESCRIPTION

Name	**Description**
config	Configuration number to set

RETURN VALUE

None

C.3.8 WDF_SetInterface()

PURPOSE

• This function is called by the Task Dispatcher when a SET INTERFACE command is received.

PROTOTYPE

```
void WDF_SetInterface(
    byte ifc,
    byte alt_set);
```

PARAMETERS

Name	Type	Input/Output
➢ ifc	byte	Input
➢ alt_set	byte	Input

DESCRIPTION

Name	Description
ifc	Interface number to set
alt_set	Alternate setting number to set

RETURN VALUE

None

C.3.9 WDF_GetInterface()

PURPOSE

• This function is called by the Task Dispatcher when a GET INTERFACE command is received.

PROTOTYPE

```
byte WDF_GetInterface(byte ifc);
```

PARAMETERS

Name	Type	Input/Output
➢ ifc	byte	Input

DESCRIPTION

Name	Description
ifc	Interface number

RETURN VALUE

Returns the number of the active alternate setting for the given interface.

C.3.10 WDF_VendorCmnd()

PURPOSE

• This function is called by the Task Dispatcher when a vendor-specific command is received.

PROTOTYPE

```
BOOL WDF_VendorCmnd(
    byte bRequest,
    word wValue,
    word wIndex,
    word wLength);
```

PARAMETERS

Name	Type	Input/Output
➢ bRequest	byte	Input
➢ wValue	word	Input
➢ wIndex	word	Input
➢ wLength	word	Input

DESCRIPTION

Name	Description
bRequest	The actual request
wValue	The request's `wValue` field
wIndex	The request's `wIndex` field
wLength	The number of bytes to transfer (if the request has a data stage)

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

C.3.11 WDF_ClearFeature()

PURPOSE

• This function is called by the Task Dispatcher when a CLEAR FEATURE command is received.

PROTOTYPE

```
BOOL WDF_ClearFeature(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

C.3.12 WDF_SetFeature()

PURPOSE

• This function is called by the Task Dispatcher when a SET FEATURE command is received.

PROTOTYPE

```
BOOL WDF_SetFeature(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

C.4 Generated DriverWizard Mass Storage Firmware API

This section describes the WinDriver USB Device generated DriverWizard mass storage firmware API for accessing the storage media on devices based on the Microchip PIC18F4550 development board.

The functions described in this section are declared in the **18F4550\include\class\msd\wdf_disk.h** header file and implemented in the generated DriverWizard **wdf_xxx_hw.c** file. This file actually contains functions implementation stubs, which need to be filled by the user according to the specific storage media used with the PIC18F4550 board. The mass storage **WinDriver\wdf\microchip\18F4550\samples\msd\sdcard.c** sample file contains a sample implementation of the storage media access functions for a Secure Digital Card (**SD Card**).

> **NOTE**
> When modifying the generated code, make sure that you comply with the USB and Mass Storage Specifications and with your hardware's specification – see note in section [12.4.3].

C.4.1 Generated Mass Storage Firmware Types

C.4.1.1 WDF_DISK_STATUS Enumeration

Enumeration of status codes for operations performed on a device's storage media:

Enum Value	Description
DISK_STATUS_SUCCESS	Status OK
DISK_INIT_COMM_FAILURE	Communication has never been established with the storage media
DISK_INIT_TIMEDOUT	Storage media initialization has timed out
DISK_TYPE_INVALID	Failed defining the storage media's type
DISK_INVALID_COMMAND	Command was not recognized by the storage media
DISK_TIMEDOUT	Storage media timed out during a read, write or erase sequence
DISK_CRC_ERROR	A CRC error occurred during a read transfer; the read data should be invalidated
DISK_DATA_REJECTED	The storage media's CRC did not match that of the data that was sent

C.4.1.2 DISK_STATE Union

Storage media state union type:

Name	**Type**	**Description**
➢	struct	
❏ is_initialized	bit field (1)	Set to 1 after a successful media initialization
➢ _byte	byte	

C.4.2 Generated Mass Storage Firmware Functions

C.4.2.1 WDF_DISK_MediaInitialize()

PURPOSE

• Initializes the device's storage media.

PROTOTYPE

```
WDF_DISK_STATUS WDF_DISK_MediaInitialize(void);
```

RETURN VALUE

Returns the result of the storage media initialization [C.4.1.1].

C.4.2.2 WDF_DISK_SectorRead()

PURPOSE

• Reads a given sector from the device's storage media.

PROTOTYPE

```
WDF_DISK_STATUS WDF_DISK_SectorRead(
    dword sector_addr,
    byte *buffer);
```

PARAMETERS

Name	Type	Input/Output
➢ sector_addr	dword	Input
➢ buffer	byte*	Output

DESCRIPTION

Name	Description
sector_addr	Sector address to read from
buffer	Pointer to a buffer to be updated with the read data

RETURN VALUE

Returns the read result [C.4.1.1].

C.4.2.3 WDF_DISK_SectorWrite()

PURPOSE

• Writes a given sector to the device's storage media.

PROTOTYPE

```
WDF_DISK_STATUS WDF_DISK_SectorWrite(
    dword sector_addr,
    byte *buffer);
```

PARAMETERS

Name	Type	Input/Output
➢ sector_addr	dword	Input
➢ buffer	byte*	Input

DESCRIPTION

Name	Description
sector_addr	Sector address to write to
buffer	Pointer to a buffer containing the data to write

RETURN VALUE

Returns the write result [C.4.1.1].

C.4.2.4 WDF_DISK_Detect()

PURPOSE

• Checks if storage media is present on the device.

PROTOTYPE

```
BOOL WDF_DISK_Detect(void);
```

RETURN VALUE

Returns TRUE if storage media is present; otherwise returns FALSE.

C.4.2.5 WDF_DISK_IsWriteProtected()

PURPOSE

• Checks if the storage media on the device is write protected.

PROTOTYPE

```
BOOL WDF_DISK_IsWriteProtected(void);
```

RETURN VALUE

Returns TRUE if the storage media is write protected; otherwise returns FALSE.

C.4.2.6 WDF_DISK_GetCapacity()

PURPOSE

• Gets the capacity of the device's storage media.

PROTOTYPE

```
void WDF_DISK_GetCapacity(
    dword *last_lba,
    dword *block_len);
```

PARAMETERS

Name	Type	Input/Output
➢ last_lba	dword*	Output
➢ block_len	dword*	Output

DESCRIPTION

Name	Description
last_lba	The storage media's last logical block address (LBA)
block_len	The storage media's block length (in bytes)

RETURN VALUE

None

Appendix D

USB Device – Philips PDIUSBD12 API Reference

D.1 Firmware Library API

This section describes the WinDriver USB Device firmware library API for the Philips PDIUSBD12 development board. The functions and general types and definitions described in this section are declared and defined (respectively) in the **d12\include\d12_lib.h** and **d12\include\types.h** header files. The functions are implemented in the generated DriverWizard **d12_lib.c** file – for registered users, or in the **d12\lib\d12_eval.lib** evaluation firmware library – for evaluation users (see section 12.3.5 for details).

> **NOTE**
> Registered users can modify the library source code. When modifying the code, make sure that you comply with the USB Specification and with your hardware's specification – see note in section 12.4.3.

D.1.1 Firmware Library Types

The data types described in this section are defined in the **d12\include\types.h** header file.

D.1.1.1 WDF_ENDPOINTS Enumeration

Enumeration of PDIUSBD12 endpoints:

Enum Value	Description
EP0_OUT	Control (endpoint 0) OUT endpoint
EP0_IN	Control (endpoint 0) IN endpoint
EP1_OUT	Generic OUT endpoint
EP1_IN	Generic IN endpoint
EP2_OUT	Main OUT endpoint
EP2_IN	Main IN endpoint

D.1.1.2 D12_MODES Enumeration

Enumeration of PDIUSBD12 modes:

Enum Value	Description
NO_ISO	None of the main endpoints are isochronous
ISO_IN	The main IN endpoint is isochronous
ISO_OUT	The main OUT endpoint is isochronous
ISO_INOUT	Both of the main endpoints are isochronous

D.1.1.3 DMA_DIRECTION Enumeration

Enumeration of Direct Memory Access (DMA) directions:

Enum Value	Description
DMA_IN	DMA direction IN – from the device memory to the PDIUSBD12 (and from there to the host)
DMA_OUT	DMA direction OUT – from the PDIUSBD12 (as received from the host) to the device's memory

D.1.2 Firmware Library Functions

The functions described in sections D.1.2.1 – D.1.2.16 are declared in the **d12\d12_lib.h** header file.

The functions described in sections D.1.2.17 – D.1.2.18 are declared in **d12\d12_io.h**. These functions provide the library's hardware abstraction layer. The default implementation of these functions is targeted at the D12-ISA (PC) Eval Kit, version 1.4, which supports connection of a PDIUSBD12-based board to an x86 PC using an ISA card. However, registered WinDriver USB Device users can modify the implementation of these functions, in the generated DriverWizard **d12_io.c** file, as well as the hardware-specific definitions in the **d12_io.h** header file, in order to support any other appropriate microcontroller.

D.1.2.1 WDF_Exit()

PURPOSE

• Exists the firmware library.

PROTOTYPE

```
void WDF_Exit(void);
```

RETURN VALUE

None

D.1.2.2 WDF_ConnectUSB()

PURPOSE

• Establishes communication between the device and the USB bus.

PROTOTYPE

```
void WDF_ConnectUSB(D12_MODES mode);
```

PARAMETERS

Name	Type	Input/Output
➢ mode	D12_MODES	Input

DESCRIPTION

Name	Description
mode	PDIUSBD12 mode [D.1.1.2]

RETURN VALUE

None

D.1.2.3 WDF_DisconnectUSB()

PURPOSE

- Disconnects communication between the device and the USB bus.

PROTOTYPE

```
void WDF_DisconnectUSB(void);
```

RETURN VALUE

None

D.1.2.4 WDF_ReconnectUSB()

PURPOSE

• Disconnects communication between the device and the USB bus (`WDF_DisconnectUSB()`) and then re-connects the communication (`WDF_ConnectUSB()`).

PROTOTYPE

```
void WDF_ReconnectUSB(D12_MODES mode);
```

PARAMETERS

Name	Type	Input/Output
➢ mode	D12_MODES	Input

DESCRIPTION

Name	Description
mode	PDIUSBD12 mode [D.1.1.2]

RETURN VALUE

None

D.1.2.5 WDF_EnableAllEP()

PURPOSE

- Enables all the device's endpoints.

PROTOTYPE

```
void WDF_EnableAllEP(void);
```

RETURN VALUE

None

D.1.2.6 WDF_DisableEP1AND2()

PURPOSE

- Disables the device's generic endpoint (EP1) and main endpoint (EP2).

PROTOTYPE

```
void WDF_DisableEP1AND2(void);
```

RETURN VALUE

None

D.1.2.7 WDF_StallEP0()

PURPOSE

- Stalls the device's control endpoint (endpoint 0).

PROTOTYPE

```
void WDF_StallEP0(void);
```

RETURN VALUE

None

D.1.2.8 WDF_EPoutFull()

PURPOSE

• Checks if a given generic or main OUT endpoint's data buffer contains data from the host.

PROTOTYPE

```
unsigned char WDF_EPoutFull(WDF_ENDPOINTS ep);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	WDF_ENDPOINTS	Input

DESCRIPTION

Name	Description
ep	The endpoint to check [D.1.1.1]

RETURN VALUE

Returns 1 if the endpoint's data buffer contains data from the host, 0 if it does not, and `GENERR` in case of an error (i.e. **`ep`** is not of type `EP1_OUT` or `EP2_OUT`).

D.1.2.9 WDF_EPinFull()

PURPOSE

• Checks if a given generic or main IN endpoint is ready to accept data from the firmware (which will later be transferred to the host).

PROTOTYPE

```
unsigned char WDF_EPinFull(WDF_ENDPOINTS ep);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	WDF_ENDPOINTS	Input

DESCRIPTION

Name	Description
ep	The endpoint to check [D.1.1.1]

RETURN VALUE

Returns 1 if the endpoint is ready to accept data, 0 if it is not, and `GENERR` in case of an error (i.e. **`ep`** is not of type `EP1_OUT` or `EP2_OUT`).

D.1.2.10 WDF_EPWrite()

PURPOSE

• Writes data to a given endpoint.

PROTOTYPE

```
unsigned char WDF_EPWrite(
    WDF_ENDPOINTS ep,
    unsigned char code *pData,
    unsigned short len);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	WDF_ENDPOINTS	Input
➢ pData	unsigned char code*	Input
➢ len	unsigned short	Input

DESCRIPTION

Name	Description
ep	The endpoint to write to [D.1.1.1]
pData	Pointer to a buffer containing the data to write
len	The number of bytes to write

RETURN VALUE

Returns 0 or `GENERR` in case of invalid parameters.

D.1.2.11 WDF_EPRead()

PURPOSE

- Reads data from a given endpoint.

PROTOTYPE

```
unsigned char WDF_EPRead(
    WDF_ENDPOINTS ep,
    unsigned char code *pData,
    unsigned short len);
```

PARAMETERS

Name	Type	Input/Output
➢ ep	WDF_ENDPOINTS	Input
➢ pData	unsigned char code*	Output
➢ len	unsigned short	Input

DESCRIPTION

Name	Description
ep	The endpoint to write to [D.1.1.1]
pData	Pointer to a buffer containing the read data
len	The number of bytes to read

RETURN VALUE

Returns 0 or `GENERR` in case of invalid parameters.

D.1.2.12 WDF_DMASetup()

PURPOSE

- Sets up a Direct Memory Access (DMA) transfer.

PROTOTYPE

```
void WDF_DMASetup(
    DMA_DIRECTION direction,
    unsigned char dmaFlags,
    void *pUserData);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢ direction	DMA_DIRECTION	Input
➢ dmaFlags	unsigned char	Input
➢ pUserData	void*	Input/Output

DESCRIPTION

Name	**Description**
direction	The direction of the DMA transfer [D.1.1.3]
dmaFlags	DMA flags. Can be any of the following: • `D12_DMASINGLE` – DMA in single mode • `D12_BURST_4` – DMA in burst 4 mode • `D12_BURST_8` – DMA in burst 8 mode • `D12_BURST_16` – DMA in burst 16 mode
pUserData	Pointer to the DMA buffer

RETURN VALUE

None

D.1.2.13 WDF_DMARunning()

PURPOSE

• Checks if there is currently an active DMA transfer.

PROTOTYPE

```
unsigned char WDF_DMARunning( void );
```

RETURN VALUE

Returns 1 if there is currently an active DMA transfer; otherwise returns 0.

D.1.2.14 WDF_DMAStop()

PURPOSE

• Stops an active DMA transfer.

PROTOTYPE

```
void WDF_DMAStop( void );
```

RETURN VALUE

None

D.1.2.15 WDF_SetLEDStatus()

PURPOSE

• Sets the status of a given LED on the PDIUSBD12's evaluation board.
This function is specific to the D12-ISA (PC) Eval Kit.

PROTOTYPE

```
void WDF_SetLEDStatus(
    unsigned char ledNum,
    unsigned char status);
```

PARAMETERS

Name	Type	Input/Output
➢ ledNum	unsigned char	Input
➢ status	unsigned char	Input

DESCRIPTION

Name	Description
ledNum	LED number
status	LED status

RETURN VALUE

None

D.1.2.16 WDF_GetKeyStatus()

PURPOSE

• Gets the status of a given key on the PDIUSBD12's evaluation board.
This function is specific to the D12-ISA (PC) Eval Kit.

PROTOTYPE

```
char WDF_GetKeyStatus(void);
```

RETURN VALUE

Returns the key status.

D.1.2.17 outportb()

PURPOSE

• Writes a byte to a given a port.

PROTOTYPE

```
void outportb(unsigned short port, unsigned char val);
```

Name	Type	Input/Output
➢ port	unsigned short	Input
➢ val	unsigned char	Input

DESCRIPTION

Name	Description
port	The port to write to
val	The byte to write

RETURN VALUE

None

D.1.2.18 inportb()

PURPOSE

- Reads a byte from a given a port.

PROTOTYPE

```
unsigned char inportb(unsigned short port);
```

Name	Type	Input/Output
➢ port	unsigned short	Input

DESCRIPTION

Name	Description
port	The port to read from

RETURN VALUE

Returns the byte that was read from the given port.

D.2 Generated DriverWizard Firmware API

This section describes the WinDriver USB Device generated DriverWizard firmware API for the Philips PDIUSBD12 development board. The functions described in this section are declared in the **d12\include\periph.h** header file and implemented in the generated DriverWizard **periph.c** file, according to the device configuration information defined in the wizard.

The firmware's entry point – `main()` in **main.c** (source code provided for registered users only) – implements a **Task Dispatcher**, which calls the `WDF_xxx()` functions declared in **periph.h** (and implemented in **periph.c**) in order to communicate with the peripheral device.

> **NOTE**
> When modifying the generated code, make sure that you comply with the USB Specification and with your hardware's specification – see note in section [12.4.3].

D.2.1 WDF_Init()

PURPOSE

• Performs user-specific device initialization and sets the device's alternate settings modes.

This function is automatically called from the firmware's `main()` function during the initialization of the firmware library.

PROTOTYPE

```
void WDF_Init(void);
```

RETURN VALUE

None

D.2.2 WDF_Uninit()

PURPOSE

• Performs user-specific device un-initialization.

This function is automatically called from the firmware's `main()` function during the un-initialization of the firmware library.

PROTOTYPE

```
void WDF_Uninit(void);
```

RETURN VALUE

None

D.2.3 WDF_SuspendChange()

PURPOSE

• This function is called by the Task Dispatcher before the device goes into suspend mode or when it returns from suspend mode.

PROTOTYPE

```
void WDF_SuspendChange(void);
```

RETURN VALUE

Returns TRUE if successful; otherwise returns FALSE.

D.2.4 WDF_Poll()

PURPOSE

• Polls the device for data.
The Task Dispatcher calls this function repeatedly.

PROTOTYPE

```
void WDF_Poll(void);
```

RETURN VALUE

None

D.2.5 WDF_BusReset()

PURPOSE

• This function is called by the Task Dispatcher when a bus reset occurs.

PROTOTYPE

```
void WDF_BusReset(void);
```

RETURN VALUE

None

D.2.6 WDF_SetConfiguration()

PURPOSE

• This function is called by the Task Dispatcher when a SET CONFIGURATION command is received.

PROTOTYPE

```
void WDF_SetConfiguration(void);
```

RETURN VALUE

None

D.2.7 WDF_SetInterface()

PURPOSE

• This function is called by the Task Dispatcher when a SET INTERFACE command is received.

PROTOTYPE

```
void WDF_SetInterface(void);
```

RETURN VALUE

None

D.2.8 WDF_GetInterface()

PURPOSE

• This function is called by the Task Dispatcher when a GET INTERFACE command is received.

PROTOTYPE

```
void WDF_GetInterface(void);
```

RETURN VALUE

None

D.2.9 WDF_VendorRequest()

PURPOSE

• This function is called by the Task Dispatcher when a vendor-specific command is received.

PROTOTYPE

```
char WDF_VendorRequest(
    unsigned char bRequest,
    unsigned short wValue,
    unsigned short wIndex,
    unsigned short wLength,
    unsigned char *pData,
    unsigned short *pwRetLen,
    unsigned char **ppRetData);
```

Parameters

Name	Type	Input/Output
➢ bRequest	unsigned char	Input
➢ wValue	unsigned short	Input
➢ wIndex	unsigned short	Input
➢ wLength	unsigned short	Input
➢ pData	unsigned char*	Input
➢ pwRetLen	unsigned short*	Output
➢ ppRetData	unsigned char**	Output

Description

Name	Description
bRequest	The actual request
wValue	The request's `wValue` field
wIndex	The request's `wIndex` field
wLength	The number of bytes to transfer (if the request has a data stage)
pData	Pointer to the data for the request's data stage, as received from the host (relevant only for OUT requests)
pwRetLen	The length (in bytes) of the returned data (`*ppRetData`)
ppRetData	Pointer to a pointer to a buffer containing the data to be sent to the host in the request's data stage (relevant only for IN requests)

Return Value

Returns zero for an illegal vendor request; otherwise returns a non-zero value.

Appendix E

USB Device – Silicon Laboratories C8051F320 and C8051F340 API Reference

NOTE
The **Silicon Laboratories C8051F320** API described in this reference can also be used to develop firmware for the **Silicon Laboratories C8051F340** development board.

E.1 Firmware Library API

This section describes the WinDriver USB Device firmware library API for the Silicon Laboratories C8051F320 development board. The functions and general types and definitions described in this section are declared and defined (respectively) in the **F320\include\wdf_silabs_lib.h** header file. The functions are implemented in the generated DriverWizard **wdf_silabs_lib.c** file – for registered users, or in the **F320\lib\wdf_silabs_f320_eval.lib** evaluation firmware library – for evaluation users (see section 12.3.5 for details).

NOTE
Registered users can modify the library source code. When modifying the code, make sure that you comply with the USB Specification and with your hardware's specification – see note in section 12.4.3.

E.1.1 wdf_silabs_lib.h Types

The APIs described in this section are defined in **F320\wdf_silabs_lib.h**.

E.1.1.1 EP_DIR Enumeration

Enumeration of endpoint directions:

Enum Value	**Description**
DIR_OUT	Direction OUT (write from the host to the device)
DIR_IN	Direction IN (read from the device to the host)

E.1.1.2 EP_TYPE Enumeration

Enumeration of endpoint types.
The endpoint's type determines the type of transfers to be performed on the endpoint – bulk, interrupt or isochronous.

Enum Value	**Description**
ISOCHRONOUS	Isochronous endpoint
BULK	Bulk endpoint
INTERRUPT	Interrupt endpoint

E.1.1.3 EP_BUFFERING Enumeration

Enumeration of endpoint buffering types:

Enum Value	**Description**
NO_BUFFERING	No buffering
DOUBLE_BUFFERING	Double buffering

E.1.1.4 EP_SPLIT Enumeration

Enumeration of endpoint's FIFO (First In First Out) buffer split modes

Enum Value	Description
NO_SPLIT	Do not split the endpoint's FIFO buffer
SPLIT	Split the endpoint's FIFO buffer

E.1.2 c8051f320.h Types and General Definitions

The APIs described in this section are defined in **F320\c8051f320.h**.

E.1.2.1 Endpoint Address Definitions

The following preprocessor definitions depict an endpoint's address (i.e. its number):

Name	Description
EP1_IN	Endpoint 1, direction IN – address 0x81
EP1_OUT	Endpoint 1, direction OUT – address 0x01
EP2_IN	Endpoint 2, direction IN – address 0x82
EP2_OUT	Endpoint 2, direction OUT – address 0x02
EP3_IN	Endpoint 3, direction IN – address 0x83
EP3_OUT	Endpoint 3, direction OUT – address 0x03

E.1.2.2 Endpoint State Definitions

The following preprocessor definitions depict an endpoint's state:

Name	Description
EP_IDLE	The endpoint is idle
EP_TX	The endpoint is transferring data
EP_ERROR	An error occurred in the endpoint
EP_HALTED	The endpoint is halted
EP_RX	The endpoint is receiving data
EP_NO_CONFIGURED	The endpoint is not configured

E.1.2.3 EP_INT_HANDLER Function Pointer

Endpoint interrupt handler function pointer type.

```
typedef void (*EP_INT_HANDLER)(PEP_STATUS);
```

E.1.2.4 EP0_COMMAND Structure

Control endpoint (Pipe 0) host command information structure type:

Name	**Type**	**Description**
bmRequestType	BYTE	Request Type: *Bit 7*: Request direction (0=Host to device - Out, 1=Device to host - In). *Bits 5-6*: Request type (0=standard, 1=class, 2=vendor, 3=reserved). *Bits 0-4*: Recipient (0=device, 1=interface, 2=endpoint,3=other).
bRequest	BYTE	The specific request
wValue	WORD	A WORD-size value that varies according to the request
wIndex	WORD	A WORD-size value that varies according to the request. This value is typically used to specify an endpoint or an interface.
wLength	WORD	The length (in bytes) of the data segment for the request – i.e. the number of bytes to transfer if there is a data stage

E.1.2.5 EP_STATUS Structure

Endpoint status information structure type, used for IN, OUT and endpoint 0 (control) requests:

Name	Type	Description
bEp	BYTE	Endpoint address [E.1.2.1]
uNumBytes	UINT	Number of bytes available for transfer
uMaxP	UINT	Maximum packet size
bEpState	BYTE	Endpoint state
pData	void*	Pointer to a data buffer used for transferring data to/from the endpoint
wData	WORD	Storage for small data packets
pfIsr	EP_INT_HANDLER	Interrupt Service Routine (ISR) [E.1.2.3]

E.1.2.6 PEP_STATUS Structure Pointer

Pointer to an `EP_STATUS` structure [E.1.2.5].

E.1.2.7 IF_STATUS Structure

Interface status structure type:

Name	Type	Description
bNumAlts	BYTE	Number of alternate settings choices for the interface
bCurrentAlt	BYTE	Current active alternate setting for the interface
bIfNumber	BYTE	Interface number

E.1.2.8 PIF_STATUS Structure Pointer

Pointer to an `IF_STATUS` structure.

E.1.3 Firmware Library Functions

The functions described in this section are declared in **F320\wdf_silabs_lib.h**.

E.1.3.1 WDF_EPINConfig()

PURPOSE

• Configures endpoints 1-3 for IN transfers.

PROTOTYPE

```
void WDF_EPINConfig(
    PEP_STATUS pEpStatus,
    BYTE address,
    EP_TYPE type,
    int maxPacketSize,
    EP_INT_HANDLER pfIsr,
    EP_BUFFERING buffering,
    EP_SPLIT splitMode);
```

PARAMETERS

Name	Type	Input/Output
➢ pEpStatus	PEP_STATUS	Output
➢ address	BYTE	Input
➢ type	EP_TYPE	Input
➢ maxPacketSize	int	Input
➢ pfIsr	EP_INT_HANDLER	Input
➢ buffering	EP_BUFFERING	Input
➢ splitMode	EP_SPLIT	Input

DESCRIPTION

Name	Description
pEpStatus	Pointer to an endpoint's status information structure [E.1.2.6]. The function updates the structure with the relevant status information.
address	Endpoint address [E.1.2.1]
type	The endpoint's transfer type [E.1.1.2]
maxPacketSize	The endpoint's maximum packet size
pfIsr	The endpoint's interrupt handler [E.1.2.3]
buffering	The endpoint's buffering type [E.1.1.3]
splitMode	The endpoint's split mode [E.1.1.4]

RETURN VALUE

None

E.1.3.2 WDF_EPOUTConfig()

PURPOSE

• Configures endpoints 1-3 for OUT transfers.

PROTOTYPE

```
void WDF_EPOUTConfig(
    PEP_STATUS pEpStatus,
    BYTE address,
    EP_TYPE type,
    int maxPacketSize,
    EP_INT_HANDLER pfIsr,
    EP_BUFFERING buffering);
```

PARAMETERS

Name	Type	Input/Output
➢ pEpStatus	PEP_STATUS	Output
➢ address	BYTE	Input
➢ type	EP_TYPE	Input
➢ maxPacketSize	int	Input
➢ pfIsr	EP_INT_HANDLER	Input
➢ buffering	EP_BUFFERING	Input

DESCRIPTION

Name	**Description**
pEpStatus	Pointer to an endpoint's status information structure [E.1.2.6]. The function updates the structure with the relevant status information.
address	Endpoint address [E.1.2.1]
type	The endpoint's transfer type [E.1.1.2]
maxPacketSize	The endpoint's maximum packet size
pfIsr	The endpoint's interrupt handler [E.1.2.3]
buffering	The endpoint's buffering type [E.1.1.3]

RETURN VALUE

None

E.1.3.3 WDF_HaltEndpoint()

PURPOSE

• Halts an endpoint.

PROTOTYPE

```
BYTE WDF_HaltEndpoint(PEP_STATUS pEpStatus);
```

PARAMETERS

Name	Type	Input/Output
➢ pEpStatus	PEP_STATUS	Input/Output

DESCRIPTION

Name	Description
pEpStatus	Pointer to an endpoint's status information structure [E.1.2.6]

RETURN VALUE

Returns the endpoint's state [E.1.2.2].

E.1.3.4 WDF_EnableEndpoint()

PURPOSE

• Enables an endpoint.

PROTOTYPE

```
BYTE WDF_EnableEndpoint(PEP_STATUS pEpStatus);
```

PARAMETERS

Name	Type	Input/Output
➢ pEpStatus	PEP_STATUS	Input/Output

DESCRIPTION

Name	Description
pEpStatus	Pointer to an endpoint's status information structure [E.1.2.6]

RETURN VALUE

Returns the endpoint's state [E.1.2.2].

E.1.3.5 WDF_SetEPByteCount()

PURPOSE

• Sets the bytes count of an endpoint's FIFO (First In First Out) buffer.
The call to this function should be preceded by a call to `WDF_FIFOWrite()` [E.1.3.10] in order to update the endpoint's FIFO buffer with the data to be transferred to the host.

PROTOTYPE

```
void WDF_SetEPByteCount(
    BYTE bEp,
    UINT bytes_count);
```

PARAMETERS

Name	Type	Input/Output
➢ bEp	BYTE	Input
➢ bytes_count	UINT	Input

DESCRIPTION

Name	Description
bEp	Endpoint address [E.1.2.1]
bytes_count	Bytes count to set

RETURN VALUE

None

E.1.3.6 WDF_GetEPByteCount()

PURPOSE

• Gets the current bytes count of an endpoint's FIFO (First In First Out) buffer.
This function should be called before calling `WDF_FIFORead()` [E.1.3.11] to read from the endpoint's FIFO buffer, in order to determine the amount of bytes to read.

PROTOTYPE

```
UINT  WDF_GetEPByteCount(BYTE  bEp);
```

PARAMETERS

Name	**Type**	**Input/Output**
➢bEp	BYTE	Input

DESCRIPTION

Name	**Description**
bEp	Endpoint address [E.1.2.1]

RETURN VALUE

Returns the endpoint's FIFO bytes count.

E.1.3.7 WDF_FIFOClear()

PURPOSE

• Empties and endpoint's FIFO (First In First Out) buffer.

PROTOTYPE

```
void WDF_FIFOClear(BYTE bEp);
```

PARAMETERS

Name	Type	Input/Output
➢ bEp	BYTE	Input

DESCRIPTION

Name	Description
bEp	Endpoint address [E.1.2.1]

RETURN VALUE

None

E.1.3.8 WDF_FIFOFull()

PURPOSE

• Checks if an endpoint's FIFO (First In First Out) buffer is completely full.

PROTOTYPE

```
BOOL WDF_FIFOFull(BYTE bEp);
```

PARAMETERS

Name	Type	Input/Output
➢ bEp	BYTE	Input

DESCRIPTION

Name	Description
bEp	Endpoint address [E.1.2.1]

RETURN VALUE

Returns TRUE if the endpoint's FIFO buffer is full; otherwise returns FALSE.

E.1.3.9 WDF_FIFOEmpty()

PURPOSE

• Checks if an endpoint's FIFO (First In First Out) buffer is empty.

PROTOTYPE

```
BOOL WDF_FIFOEmpty(BYTE bEp);
```

PARAMETERS

Name	Type	Input/Output
➢ bEp	BYTE	Input

DESCRIPTION

Name	Description
bEp	Endpoint address [E.1.2.1]

RETURN VALUE

Returns TRUE if the endpoint's FIFO buffer is empty; otherwise returns FALSE.

E.1.3.10 WDF_FIFOWrite()

PURPOSE

• Writes data to an endpoint's FIFO (First In First Out) buffer.
The call to this function should be followed by a call to
`WDF_SetEPByteCount()` [E.1.3.5].

PROTOTYPE

```
void WDF_FIFOWrite(
    BYTE bEp,
    UINT uNumBytes,
    BYTE *pData);
```

PARAMETERS

Name	Type	Input/Output
➢ bEp	BYTE	Input
➢ pData	BYTE*	Input
➢ uNumBytes	UINT	Input

DESCRIPTION

Name	Description
bEp	Endpoint address [E.1.2.1]
pData	Data buffer to write
uNumBytes	Number of bytes to write

RETURN VALUE

None

E.1.3.11 WDF_FIFORead()

PURPOSE

• Reads data from an endpoint's FIFO (First In First Out) buffer.
The call to this function should be preceded
by a call to `WDF_GetEPByteCount()` [E.1.3.6] in order to determine the amount of bytes to read.

PROTOTYPE

```
void WDF_FIFORead(
    BYTE bEp,
    UINT uNumBytes,
    BYTE *pData);
```

PARAMETERS

Name	Type	Input/Output
➢ bEp	BYTE	Input
➢ pData	BYTE*	Output
➢ uNumBytes	UINT	Input

DESCRIPTION

Name	Description
bEp	Endpoint address [E.1.2.1]
pData	Buffer to hold the read data
uNumBytes	Number of bytes to read from the FIFO buffer

RETURN VALUE

None

E.1.3.12 WDF_GetEPStatus()

PURPOSE

- Gets an endpoint's status information.

PROTOTYPE

```
PEP_STATUS WDF_GetEPStatus(BYTE bEp);
```

PARAMETERS

Name	Type	Input/Output
➢ bEp	BYTE	Input

DESCRIPTION

Name	Description
bEp	Endpoint address [E.1.2.1]

RETURN VALUE

Returns a pointer to a structure that holds the endpoint's status information [E.1.2.6].

E.2 Generated DriverWizard Firmware API

This section describes the WinDriver USB Device generated DriverWizard firmware API for the Silicon Laboratories C8051F320 development board. The functions described in this section are declared in the **F320\include\periph.h** header file and implemented in the generated DriverWizard **periph.c** file, according to the device configuration information defined in the wizard.

NOTE
When modifying the generated code, make sure that you comply with the USB Specification and with your hardware's specification – see note in section [12.4.3].

E.2.1 WDF_USBReset()

PURPOSE

• Initializes the device status information to zero and resets all endpoints.

PROTOTYPE

```
void WDF_USBReset(void);
```

RETURN VALUE

None

E.2.2 WDF_SetAddressRequest()

PURPOSE

• Handles a SET ADDRESS request.

PROTOTYPE

```
void WDF_SetAddressRequest(void);
```

RETURN VALUE

None

E.2.3 WDF_SetFeatureRequest()

PURPOSE

• Handles a SET ADDRESS request.

PROTOTYPE

```
void WDF_SetFeatureRequest(void);
```

RETURN VALUE

None

E.2.4 WDF_ClearFeatureRequest()

PURPOSE

• Handles a CLEAR FEATURE request.

PROTOTYPE

```
void WDF_ClearFeatureRequest(void);
```

RETURN VALUE

None

E.2.5 WDF_SetConfigurationRequest()

PURPOSE

• Handles a SET CONFIGURATION request.

PROTOTYPE

```
void WDF_SetConfigurationRequest(void);
```

RETURN VALUE

None

E.2.6 WDF_SetDescriptorRequest()

PURPOSE

• Handles a SET DESCRIPTOR request.

PROTOTYPE

```
void WDF_SetDescriptorRequest(void);
```

RETURN VALUE

None

E.2.7 WDF_SetInterfaceRequest()

PURPOSE

• Handles a SET INTERFACE request.

PROTOTYPE

```
void WDF_SetInterfaceRequest(void);
```

RETURN VALUE

None

E.2.8 WDF_GetStatusRequest()

PURPOSE

• Handles a GET STATUS request.

PROTOTYPE

```
void WDF_GetStatusRequest(void);
```

RETURN VALUE

None

E.2.9 WDF_GetDescriptorRequest()

PURPOSE

• Handles a GET DESCRIPTOR request.

PROTOTYPE

```
void WDF_GetDescriptorRequest(void);
```

RETURN VALUE

None

E.2.10 WDF_GetConfigurationRequest()

PURPOSE

• Handles a GET CONFIGURATION request.

PROTOTYPE

```
void WDF_GetConfigurationRequest(void);
```

RETURN VALUE

None

E.2.11 WDF_GetInterfaceRequest()

PURPOSE

• Handles a GET INTERFACE request.

PROTOTYPE

```
void WDF_GetInterfaceRequest(void);
```

RETURN VALUE

None

Appendix F

Troubleshooting and Support

Please refer to `http://www.jungo.com/support/support_windriver.html` for additional resources for developers, including:

- Technical documents
- FAQs
- Samples
- Quick start guides

Appendix G

Evaluation Version Limitations

G.1 Windows 98/Me/2000/XP/Server 2003

- Each time WinDriver is activated, an **Unregistered** message appears.
- When using DriverWizard, a dialogue box with a message stating that an evaluation version is being run appears on every interaction with the hardware.
- DriverWizard [5]:
  - Each time DriverWizard is activated, an **Unregistered** message appears.
  - An evaluation message is displayed on every interaction with the hardware using DriverWizard.
- WinDriver will function for only 30 days after the original installation.

G.2 Windows CE

- Each time WinDriver is activated, an **Unregistered** message appears.
- The WinDriver CE Kernel (**windrvr6.dll**) will operate for no more than 60 minutes at a time.
- DriverWizard [5] (used on a host Windows 2000/XP/Server 2003 PC):
  - Each time DriverWizard is activated, an **Unregistered** message appears.
  - An evaluation message is displayed on every interaction with the hardware using DriverWizard.

- WinDriver CE emulation on Windows 2000/XP/Server 2003 will stop working after 30 days.

G.3 Linux

- Each time WinDriver is activated, an **Unregistered** message appears.
- DriverWizard [5]:
  - Each time DriverWizard is activated, an **Unregistered** message appears.
  - An evaluation message is displayed on every interaction with the hardware using DriverWizard.
- WinDriver's kernel module will work for no more then 60 minutes at a time.

  In order to continue working, the WinDriver kernel module must be reloaded (remove and insert the module) using the following commands:

  To remove:
  ```
  /sbin/rmmod windrvr6
  ```

  To insert:
  ```
  /sbin/modprobe windrvr6
  ```

Appendix H

Purchasing WinDriver

Fill in the order form found in **Start | WinDriver | Order Form** on your Windows start menu, and send it to Jungo via email, fax or mail (see details below).

Your WinDriver package will be sent to you via Fedex or standard postal mail. The WinDriver license string will be emailed to you immediately.

EMAIL

Support: support@jungo.com

Sales: sales@jungo.com

PHONE/FAX

Phone:

USA (Toll-Free): 1-877-514-0537

Worldwide: +972-9-8859365

Fax:

USA (Toll-Free): 1-877-514-0538

Worldwide: +972-9-8859366

WEB:

`http://www.jungo.com`

POSTAL ADDRESS

Jungo Ltd.

P.O.Box 8493

Netanya 42504

ISRAEL

Appendix I

Distributing Your Driver – Legal Issues

*WinDriver is licensed per-seat. The WinDriver license allows one developer on a single computer to develop an unlimited number of device drivers, and to freely distribute the created drivers without royalties, as outlined in the license agreement in the **WinDriver/docs/license.pdf** file.*

Appendix J

Additional Documentation

Updated Manuals

The most updated WinDriver user manuals can be found on Jungo's site at:
`http://www.jungo.com/support/support_windriver.html`.

Version History

If you wish to view WinDriver version history, please refer to `http://www.jungo.com/wdver.html`. Here you will be able to view a list of all new features, enhancements and fixes which have been added in each WinDriver version.

Technical Documents

For additional information, you may refer to the Technical Documents database on our site at:
`http://www.jungo.com/support/tech_docs_indexes/main_index.html`.
The Technical Documents database includes detailed descriptions of WinDriver's features, utilities and APIs and their correct usage, troubleshooting of common problems, useful tips and answers to frequently asked questions.